

# Digital Design & Computer Arch.

## Lecture 15a: Out-of-Order Execution

Prof. Onur Mutlu

ETH Zürich

Spring 2020

9 April 2020

# Required Readings

---

## ■ This week

- Out-of-order execution
  - H&H, Chapter 7.8-7.9
- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

## ■ Optional

- Kessler, “**The Alpha 21264 Microprocessor**,” IEEE Micro 1999.

## ■ Next Week

- McFarling, “**Combining Branch Predictors**,” DEC WRL Technical Report, 1993.



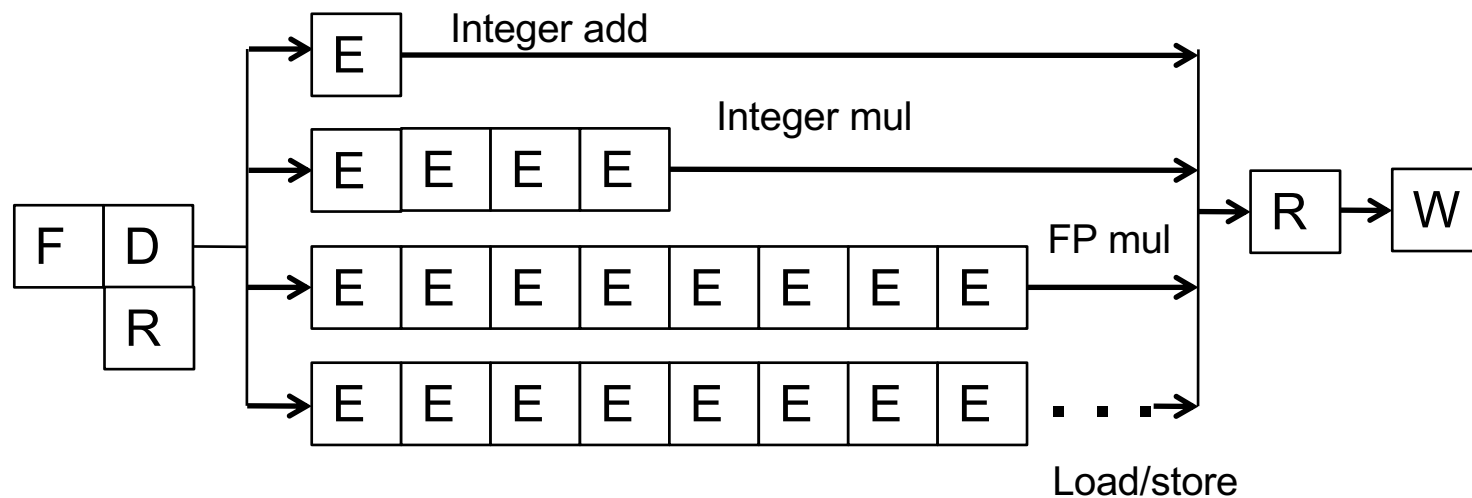
# Agenda for Today & Next Few Lectures

---

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms

# Review: In-Order Pipeline with Reorder Buffer

- **Decode (D)**: Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction (send to functional unit)
- **Execute (E)**: Instructions can complete out-of-order
- **Completion (R)**: Write result to **reorder buffer**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**




# Recall: Data Dependence Types

---

True (flow) dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW) -- **True**

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR) -- **Anti**

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write  
(WAW) -- **Output**

# Remember: Register Renaming with a Reorder Buffer

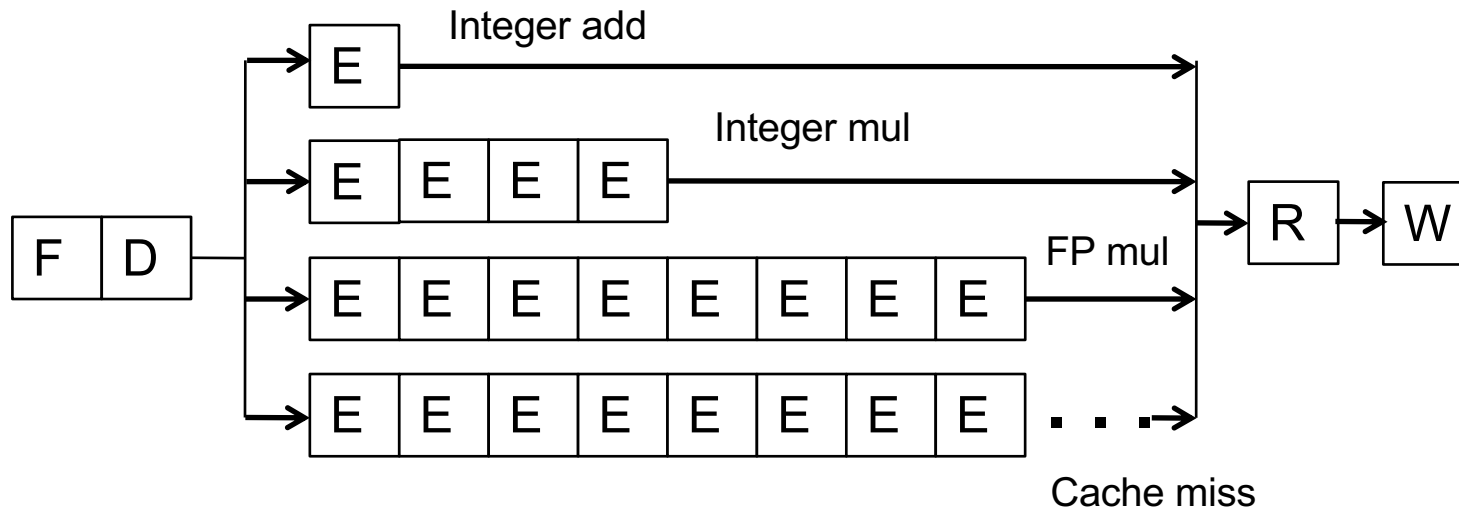
---

- Output and anti dependencies are **not true dependencies**
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register
- This eliminates anti and output dependencies
  - Gives the illusion that there are a large number of registers

# Out-of-Order Execution (Dynamic Instruction Scheduling)

# An In-order Pipeline

---



- Dispatch: Act of sending an instruction to a functional unit
- Renaming with ROB eliminates stalls due to false dependencies
- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

# Can We Do Better?

---

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R4 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

```
LD   R3 ← R1 (0)
ADD  R3 ← R3, R1
ADD  R4 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed
- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture

# Preventing Dispatch Stalls

---

- Problem: **in-order** dispatch (scheduling, or execution)
- Solution: **out-of-order** dispatch (scheduling, or execution)
- Actually, we have seen the basic idea before:
  - **Dataflow**: “fire” an instruction only when its inputs are ready
  - We will use similar principles, but not expose it in the ISA
- Aside: Any other way to prevent dispatch stalls?
  1. Compile-time instruction scheduling/reordering
  2. Value prediction
  3. Fine-grained multithreading



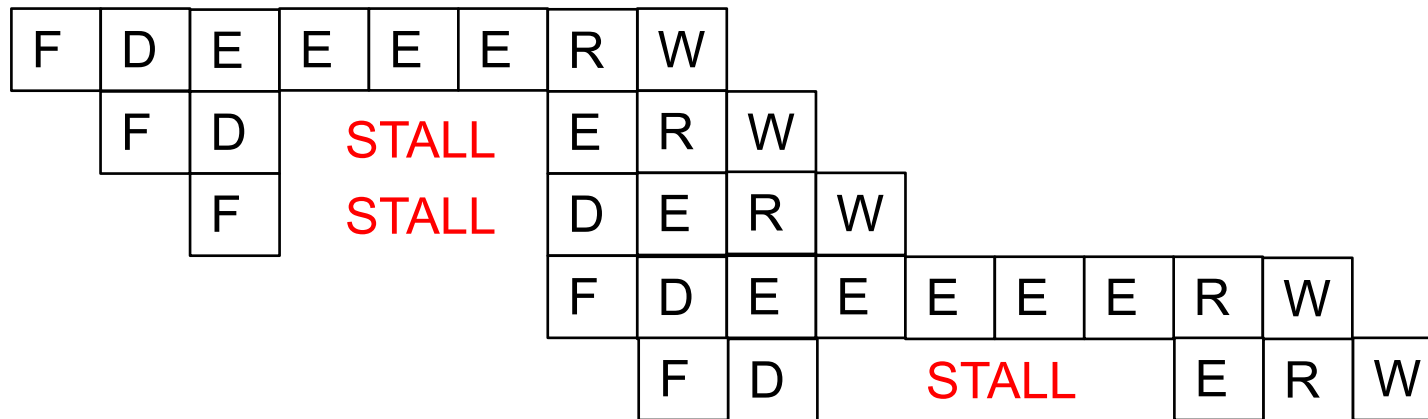
# Out-of-order Execution (Dynamic Scheduling)

---

- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
  - Rest areas for dependent instructions: Reservation stations
- Monitor the source “values” of each instruction in the resting area
- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation

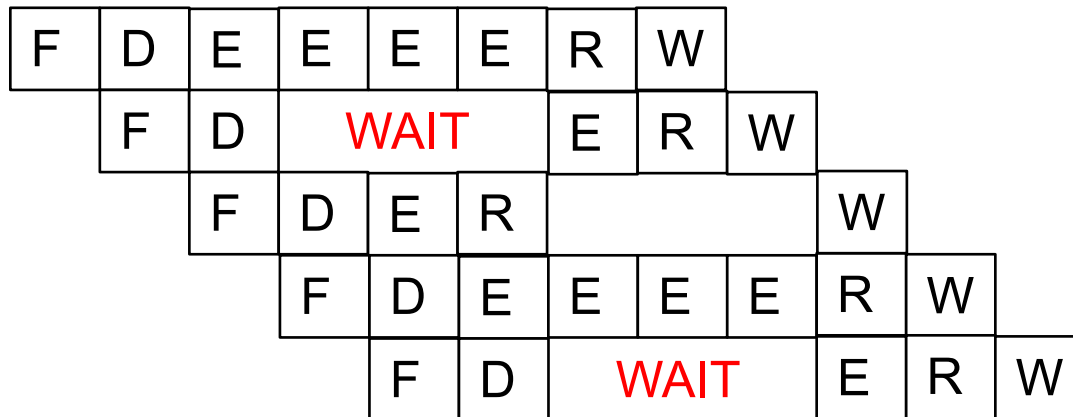
# In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



IMUL R3  $\leftarrow$  R1, R2  
 ADD R3  $\leftarrow$  R3, R1  
 ADD R1  $\leftarrow$  R6, R7  
 IMUL R5  $\leftarrow$  R6, R8  
 ADD R7  $\leftarrow$  R3, R5

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles

# Enabling OoO Execution

---

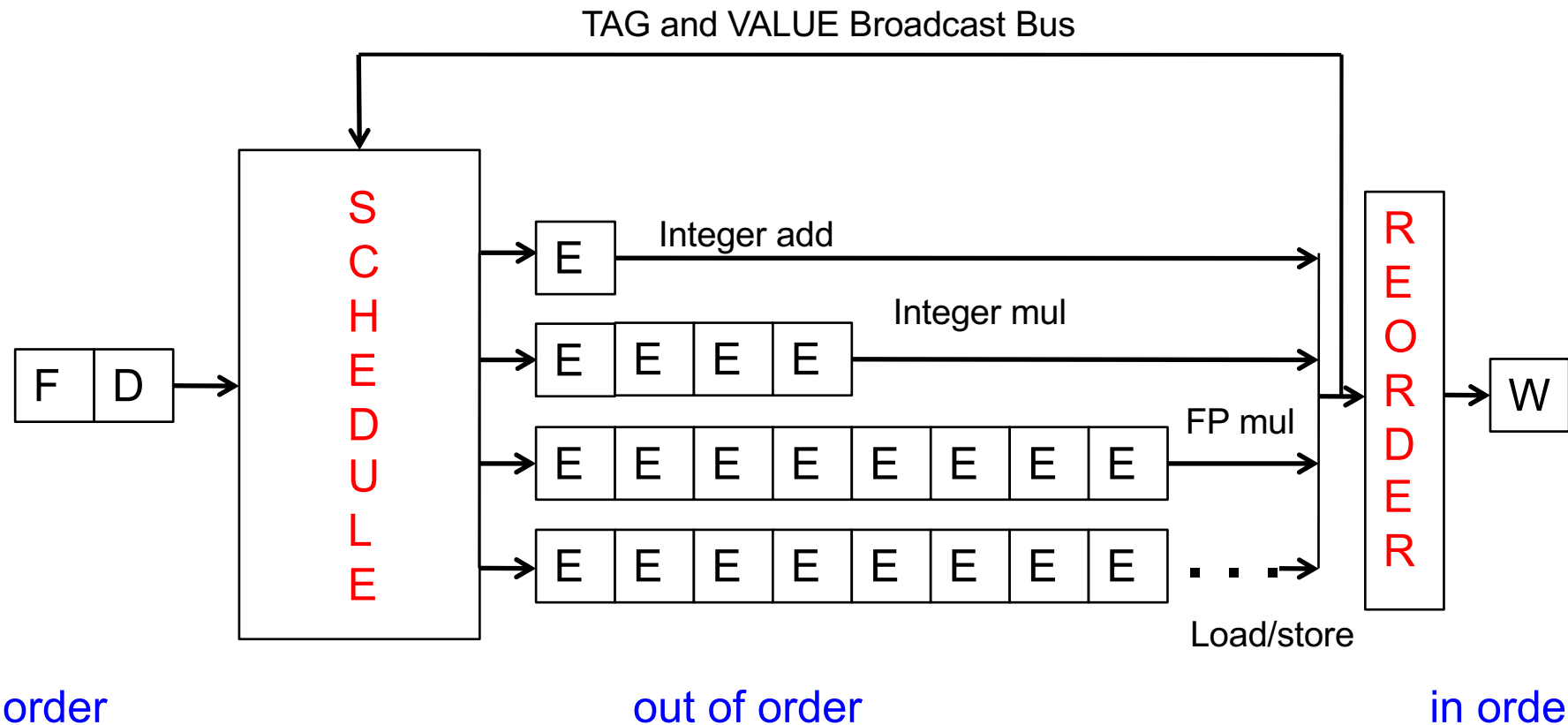
1. Need to link the consumer of a value to the producer
  - ❑ Register renaming: Associate a “tag” with each data value
2. Need to buffer instructions until they are ready to execute
  - ❑ Insert instruction into reservation stations after renaming
3. Instructions need to keep track of readiness of source values
  - ❑ Broadcast the “tag” when the value is produced
  - ❑ Instructions compare their “source tags” to the broadcast tag  
→ if match, source value becomes ready
4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
  - ❑ Instruction wakes up if all sources are ready
  - ❑ If multiple instructions are awake, need to select one per FU

# Tomasulo's Algorithm for OoO Execution

---

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - **Read:** Tomasulo, “**An Efficient Algorithm for Exploiting Multiple Arithmetic Units,**” IBM Journal of R&D, Jan. 1967.
- What is the major difference today?
  - **Precise exceptions**
  - Provided by
    - Patt, Hwu, Shebanow, “**HPS, a new microarchitecture: rationale and introduction,**” MICRO 1985.
    - Patt et al., “**Critical issues regarding HPS, a high performance microarchitecture,**” MICRO 1985.
- OoO variants are used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

# Two Humps in a Modern Pipeline



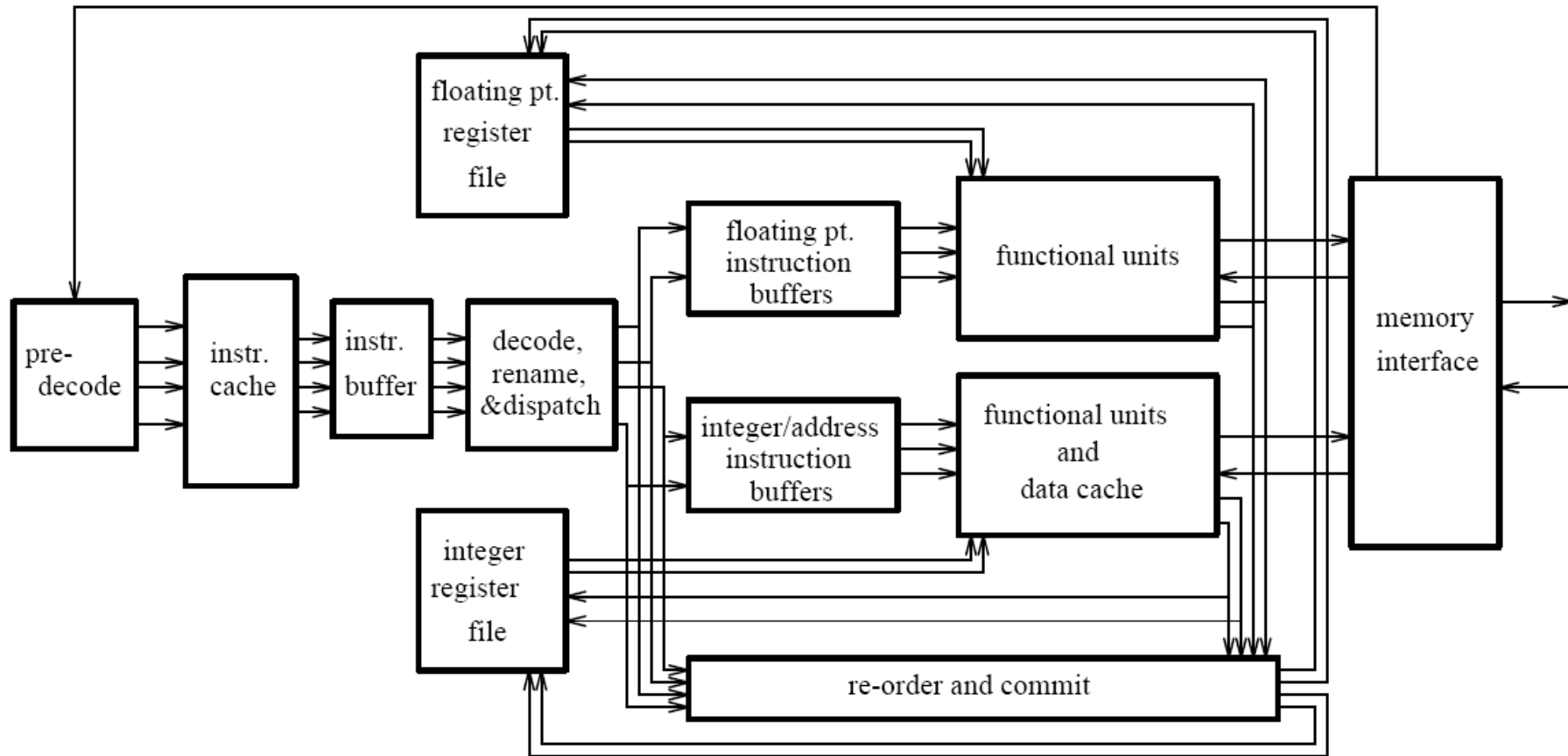
- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

# Two Humps in a Modern Pipeline

---

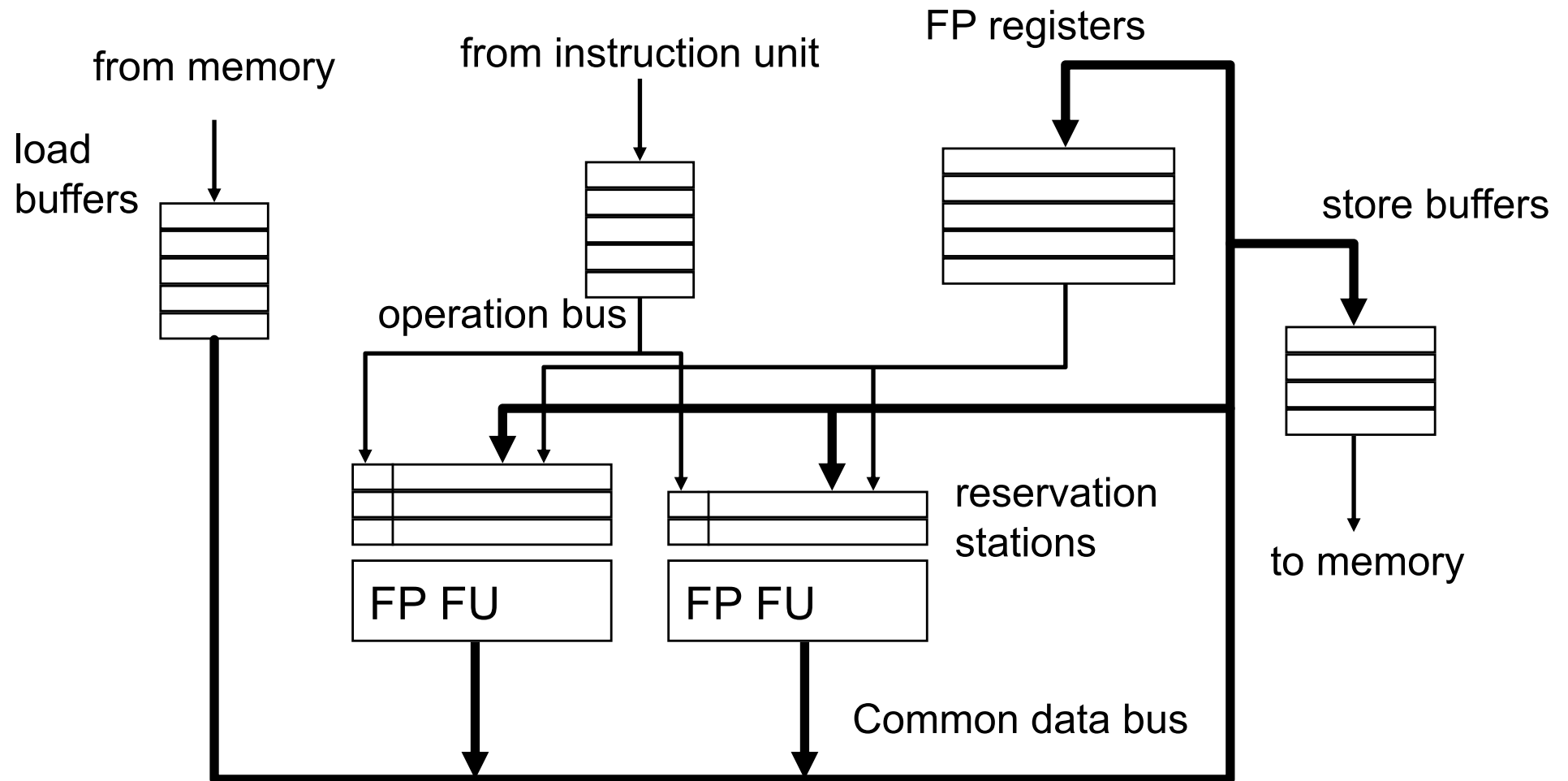


# General Organization of an OOO Processor



- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proc. IEEE, Dec. 1995.

# Tomasulo's Machine: IBM 360/91





# Recall Once More: Register Renaming

---

- Output and anti dependencies are not true dependencies
  - ❑ WHY? The same register refers to values that have nothing to do with each other
  - ❑ **They exist because not enough register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reservation station entry that will hold the register's value
  - ❑ Register ID → RS entry ID
  - ❑ Architectural register ID → Physical register ID
  - ❑ After renaming, RS entry ID used to refer to the register
- This eliminates anti- and output- dependencies
  - ❑ Approximates the performance effect of a large number of registers even though ISA has a small number

# Tomasulo's Algorithm: Renaming

---

- Register rename table (register alias table)

	tag	value	valid?
R0			1
R1			1
R2			1
R3			1
R4			1
R5			1
R6			1
R7			1
R8			1
R9			1

# Tomasulo's Algorithm

---

- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!

# An Exercise

---

MUL R3  $\leftarrow$  R1, R2  
ADD R5  $\leftarrow$  R3, R4  
ADD R7  $\leftarrow$  R2, R6  
ADD R10  $\leftarrow$  R8, R9  
MUL R11  $\leftarrow$  R7, R10  
ADD R5  $\leftarrow$  R5, R11



- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine imprecise exceptions (full forwarding)

# Exercise Continued

Pipeline structure

MUL R1, R2, → R3  
ADD R3, R4 → R5  
ADD R2, R6 → R7  
ADD R8, R9 → R10  
MUL R7, R10 → R11  
ADD R5, R11, → R5

F D E W

↓  
can take  
multiple  
cycles

MUL takes 6 cycles

ADD takes 4 cycles

How many cycles total w/o data forwarding?

" " " " w/ " " ?



# Exercise Continued

```

F D 1 2 3 4 5 6 W
  F D - - - - - D 1 2 3 4 W
    F - - - - - D 1 2 3 4 W
      F D 1 2 3 4 W
        F D - - - - D 1 2 3 4 5 6 W
          F - - - - D
                                D 1 2 3 4 W
  
```

Execution timeline w/ scoreboarding

31 cycles

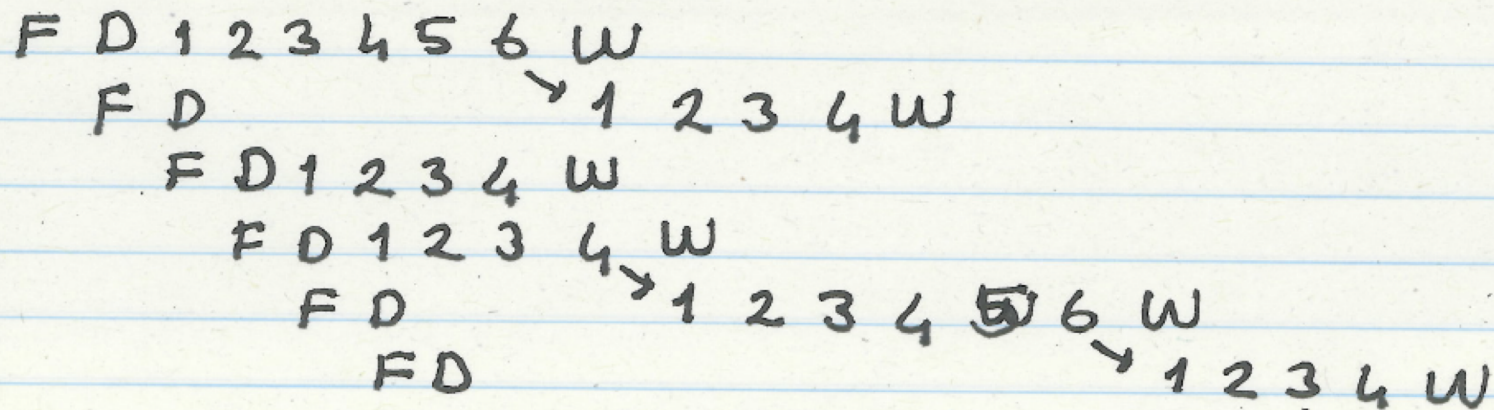
```

F D 1 2 3 4 5 6 W
  F D
    F
      E, 1 2 3 4 W
        D 1 2 3 4 W
          F D 1 2 3 4 W
            F D
              1 2 3 4 5 6 W
                D
                  1 2 3 4 W
  
```

25 cycles

# Exercise Continued

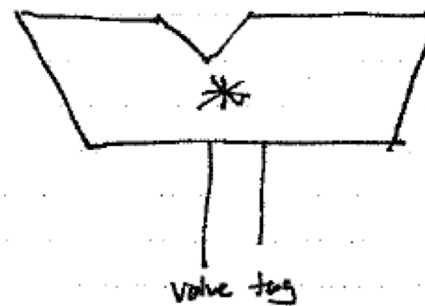
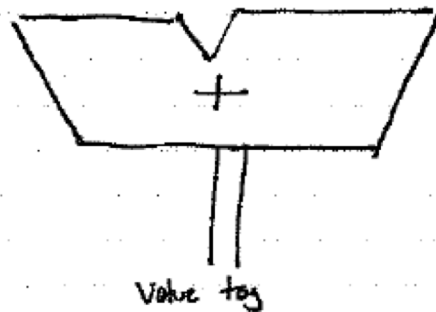
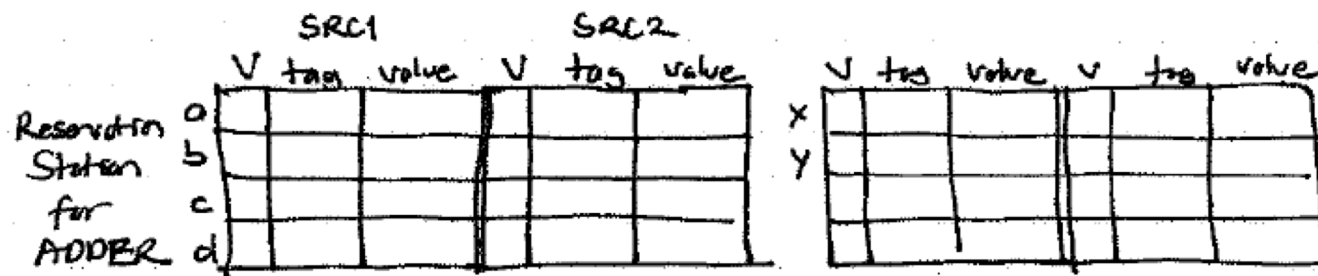
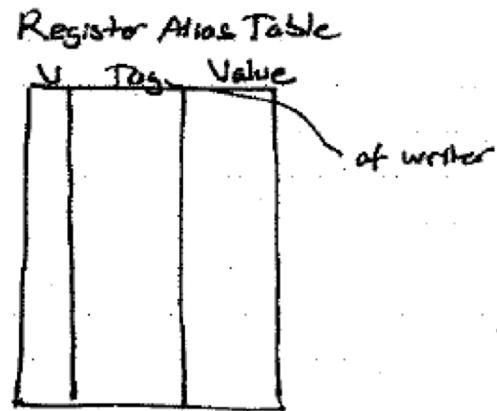
MUL R3  $\leftarrow$  R1, R2  
ADD R5  $\leftarrow$  R3, R4  
ADD R7  $\leftarrow$  R2, R6  
ADD R10  $\leftarrow$  R8, R9  
MUL R11  $\leftarrow$  R7, R10  
ADD R5  $\leftarrow$  R5, R11



Tomasulo's algorithm + full forwarding

20 cycles

# How It Works



Assume  
adder &  
multiplier have  
separate  
buses



# Our First OoO Machine Simulation

## Program We Will Simulate

```
MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

Initially:

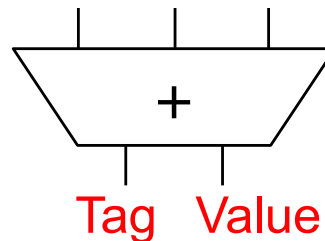
1. RS's are all Invalid (Empty)
2. All Registers are Valid

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

## Register Alias Table

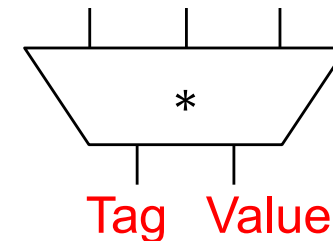
### RS for ADD Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



### RS for MUL Unit

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



ADD and MUL Execution Units  
have separate buses

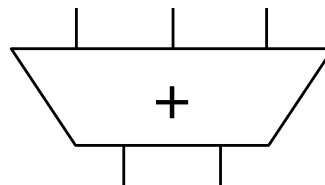
# Cycle 0

Cycle

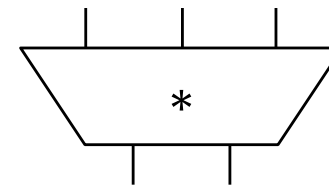
MUL R1, R2 → R3  
ADD R3, R4 → R5  
ADD R2, R6 → R7  
ADD R8, R9 → R10  
MUL R7, R10 → R11  
ADD R5, R11 → R5

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



# Cycle 1

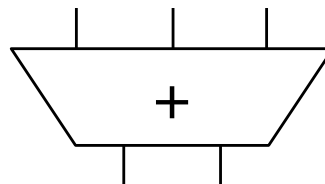
Cycle 1

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

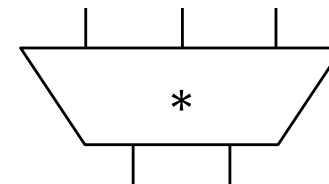
F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		3
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x						
y						
z						
t						



# Cycle 2

MUL gets decoded and allocated into RS x

Step 1: Check if reservation station available. Yes: x

Step 2: Access the Register Alias Table

Step 3: Put source registers into reservation station x.

Step 4: Rename destination register R3 → x

R3 is now renamed to x.

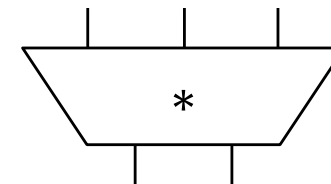
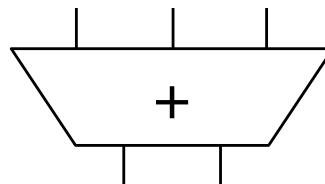
Its new value will be produced by the reservation station that is identified by tag x.

				Cycle	1	2
MUL	R1, R2	→	R3	F	D	
ADD	R3, R4	→	R5		F	
ADD	R2, R6	→	R7			
ADD	R8, R9	→	R10			
MUL	R7, R10	→	R11			
ADD	R5, R11	→	R5			

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	1		5
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a						
b						
c						
d						

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x		~			~	
y						
z						
t						



MUL in RS x is ready to execute in the next cycle!

# Cycle 3

1. MUL in RS x starts executing

2. ADD gets decoded and allocated into RS a

Cycle 1 2 3

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

F D E<sub>1</sub>  
 F D  
 F

Check readiness (Both sources ready?) → Wakeup

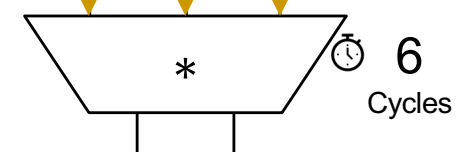
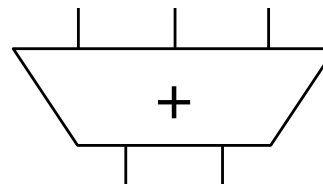
Ready → Dispatch the instruction to the MUL unit

Same Steps 1-4 for ADD... Rename R5 → a

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	1		7
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a					~	
b						
c						
d						

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y						
z						
t						



ADD in RS a cannot execute in the next cycle: one source is not valid

# Cycle 4

Cycle 1 2 3 4

ADD in RS a waits because one source is not valid.

Rename R7 → b

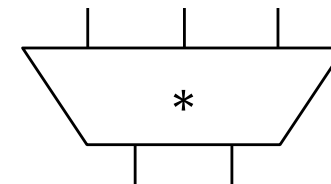
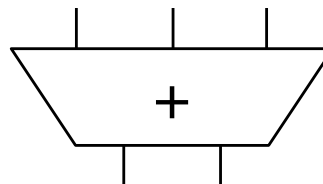
MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

F D E<sub>1</sub> E<sub>2</sub>  
 F D -  
 F D  
 F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	1		10
R11	1		11

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b		~			~	
c						
d						

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y						
z						
t						



ADD in RS b is ready to execute in the next cycle!

It will be executed out of order in the next cycle.

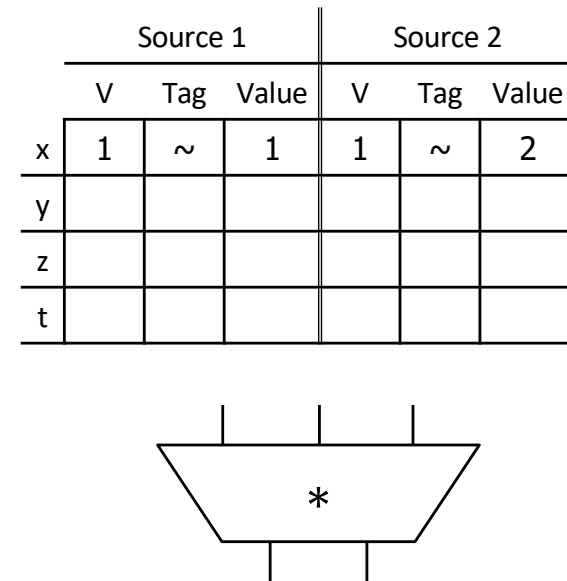
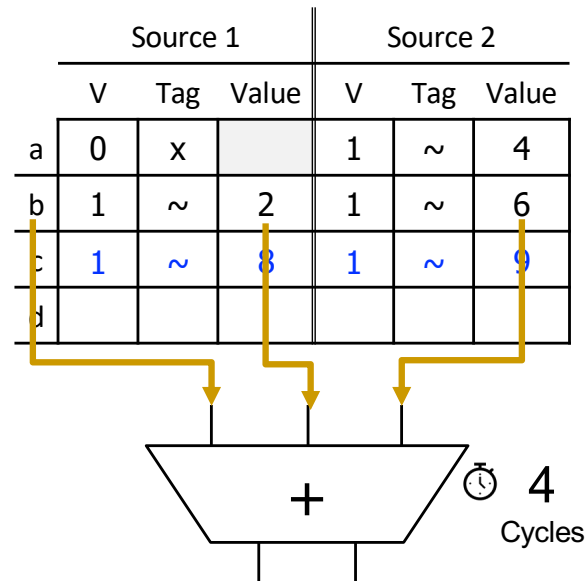
# Cycle 5

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle 1 2 3 4 5

F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub>  
 F D - -  
 F D E<sub>1</sub>  
 F D  
 F

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	1		11



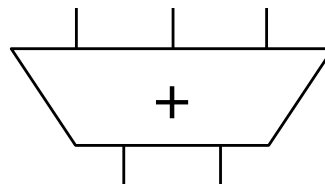
ADD in RS c is ready to execute in the next cycle!

# Cycle 6

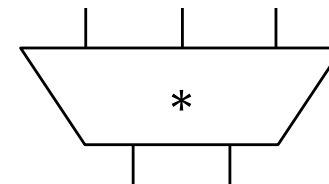
	Cycle	1	2	3	4	5	6
MUL R1, R2 → R3	F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	
ADD R3, R4 → R5		F	D	-	-	-	
ADD R2, R6 → R7				F	D	E <sub>1</sub>	
ADD R8, R9 → R10					F	D	
MUL R7, R10 → R11						F	
ADD R5, R11 → R5							

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	a	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d						



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						





# Cycle 7

All 6 instructions are now decoded and renamed

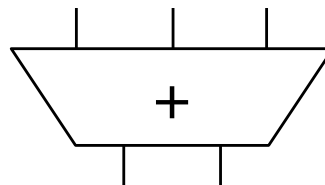
Note what happened to R5!

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

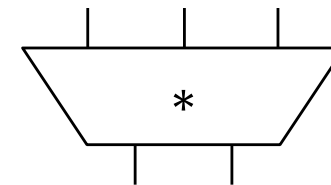
Cycle	1	2	3	4	5	6	7
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	
	F	D	-	-	-	-	
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	
			F	D	E <sub>1</sub>	E <sub>2</sub>	
				F	D	-	
					F	D	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

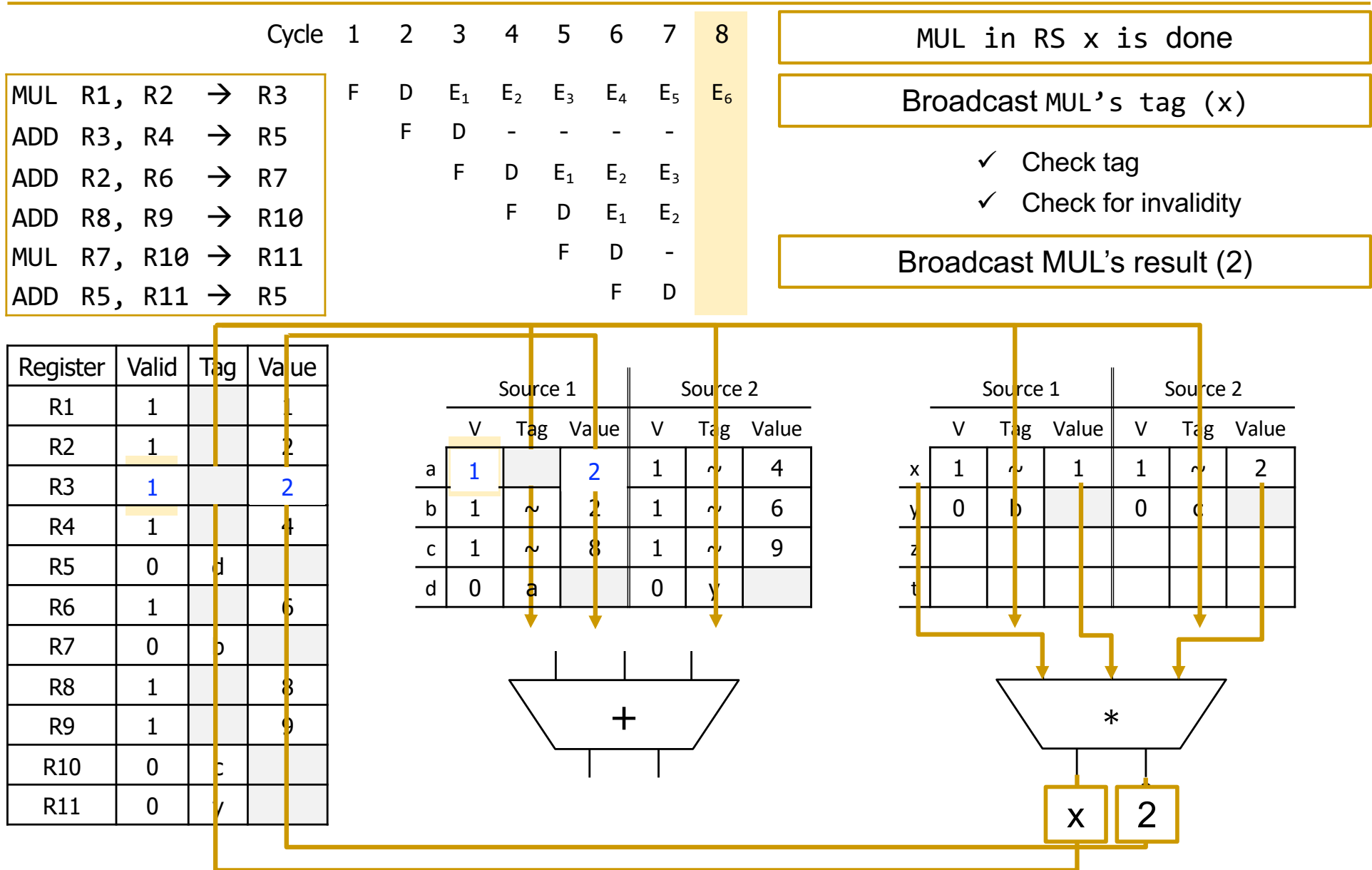
	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



# Cycle 8 (First Slide)



ADD in RS a is ready to execute in the next cycle!

# Cycle 8 (Second Slide)

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle	1	2	3	4	5	6	7	8
F		D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>
		F	D	-	-	-	-	-
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>
				F	D	E <sub>1</sub>	E <sub>2</sub>	
					F	D	-	
						F	D	

ADD in RS b is also done

Broadcast ADD's tag (b)

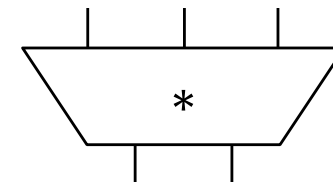
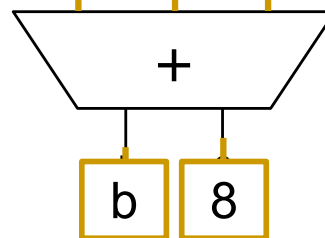
- ✓ Check tag
- ✓ Check for invalidity

Broadcast ADD's result (8)

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1		8	0	c	
z						
t						



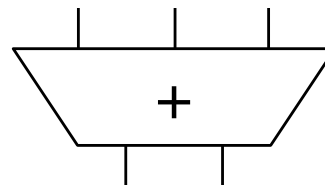
MUL in RS y is still NOT ready to execute in the next cycle!

# Cycle 8 (Third Slide)

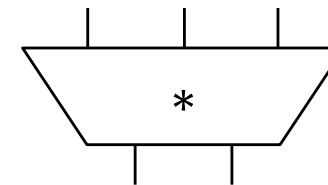
	Cycle	1	2	3	4	5	6	7	8
MUL R1, R2 → R3	F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	
ADD R3, R4 → R5		F	D	-	-	-	-	-	
ADD R2, R6 → R7			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	
ADD R8, R9 → R10				F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	
MUL R7, R10 → R11					F	D	-	-	
ADD R5, R11 → R5						F	D	-	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	0	c	
z						
t						



We did not cover the following slides.  
They are for your preparation for the  
next lecture.

# Cycle 9

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

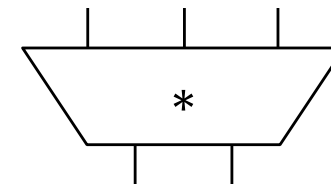
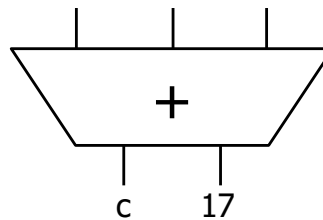
Cycle	1	2	3	4	5	6	7	8	9
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W	
	F	D	-	-	-	-	-	E <sub>1</sub>	
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W	
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	
				F	D	-	-	-	
					F	D	-	-	

Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



MUL in RS y is ready to execute in the next cycle!

# Cycle 10

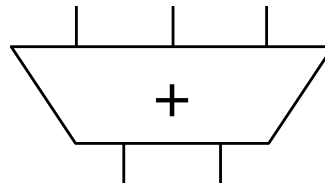
MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8 9 10

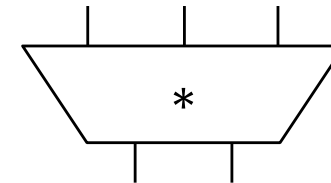
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W	
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W	
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W
				F	D	-	-	-	E <sub>1</sub>
					F	D	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



# Cycle 11

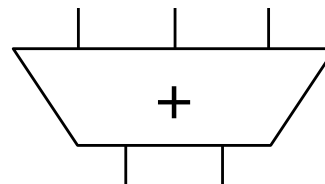
MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8 9 10 11

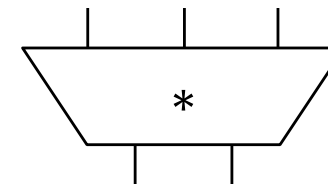
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W		
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W		
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W	
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>
					F	D	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						





# Cycle 12

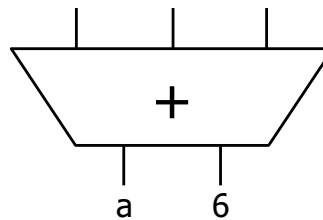
MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W				
		F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W			
				F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W		
					F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>
						F	D	-	-	-	-	-

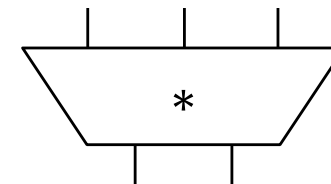
← Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



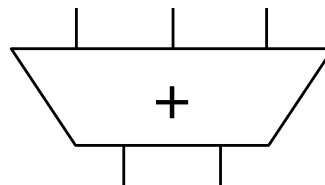
# Cycle 13

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

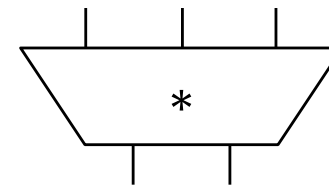
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W					
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W	
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W					
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W				
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	
					F	D	-	-	-	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



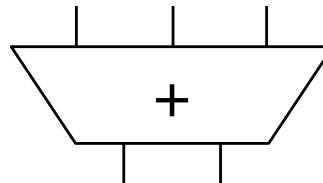
# Cycle 14

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

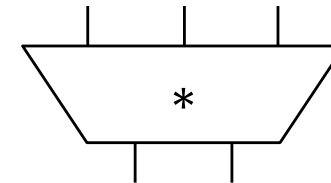
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W						
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W		
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W						
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W					
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	
					F	D	-	-	-	-	-	-	-	-

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



# Cycle 15

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

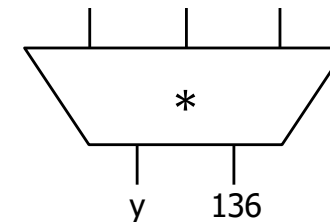
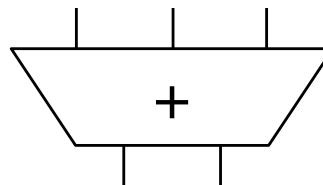
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W							
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W			
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W							
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W						
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	-
					F	D	-	-	-	-	-	-	-	-	-

Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



ADD in RS d is ready to execute in the next cycle!

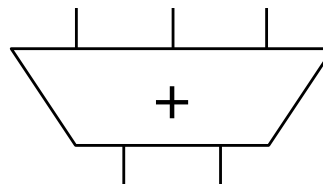
# Cycle 16

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

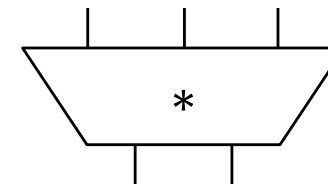
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W								
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W				
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W								
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W							
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W	
					F	D	-	-	-	-	-	-	-	-	-	E <sub>1</sub>

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



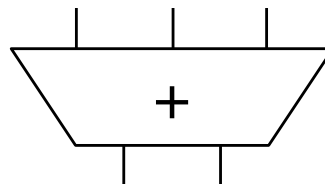
# Cycle 17

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

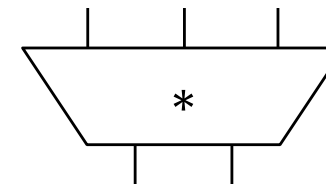
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W									
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W					
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W									
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W								
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W		
					F	D	-	-	-	-	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



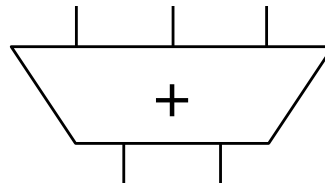
# Cycle 18

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

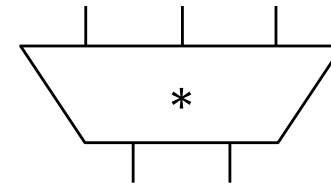
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W										
	F	D	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W						
		F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W										
			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	W									
				F	D	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	W			
					F	D	-	-	-	-	-	-	-	-	-	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	0	d	
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



# Cycle 19

MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

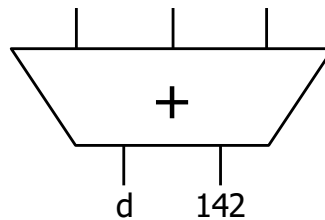
Cycle 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> E<sub>5</sub> E<sub>6</sub> W  
 F D - - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> E<sub>5</sub> E<sub>6</sub> W  
 F D - - - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub>

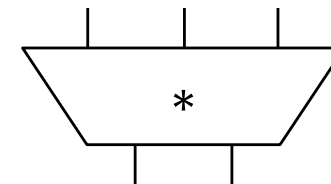
Broadcast and Update

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						





# Cycle 20

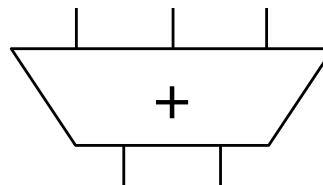
MUL R1, R2 → R3  
 ADD R3, R4 → R5  
 ADD R2, R6 → R7  
 ADD R8, R9 → R10  
 MUL R7, R10 → R11  
 ADD R5, R11 → R5

Cycle 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

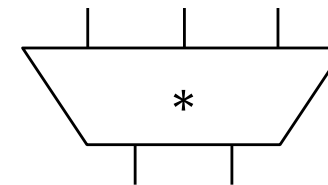
F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> E<sub>5</sub> E<sub>6</sub> W  
 F D - - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W  
 F D - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> E<sub>5</sub> E<sub>6</sub> W  
 F D - - - - - E<sub>1</sub> E<sub>2</sub> E<sub>3</sub> E<sub>4</sub> W

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	1		2
R4	1		4
R5	1		142
R6	1		6
R7	1		8
R8	1		8
R9	1		9
R10	1		17
R11	1		136

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	1	~	2	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	1	~	6	1	~	136



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	1	~	8	1	~	17
z						
t						



# Digital Design & Computer Arch.

## Lecture 15a: Out-of-Order Execution

Prof. Onur Mutlu

ETH Zürich

Spring 2020

9 April 2020

We did not cover the following slides.  
They are for your preparation for the  
next lecture.

# Some Questions

---

- What is needed in hardware to perform tag broadcast and value capture?
  - make a value valid
  - wake up an instruction
- Does the tag have to be the ID of the Reservation Station Entry?
- What can potentially become the critical path?
  - Tag broadcast → value capture → instruction wake up
- How can you reduce the potential critical paths?

# Dataflow Graph for Our Example

---

MUL R3  $\leftarrow$  R1, R2

ADD R5  $\leftarrow$  R3, R4

ADD R7  $\leftarrow$  R2, R6

ADD R10  $\leftarrow$  R8, R9

MUL R11  $\leftarrow$  R7, R10

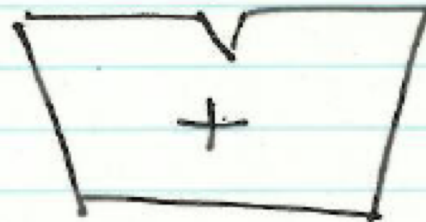
ADD R5  $\leftarrow$  R5, R11

# State of RAT and RS in Cycle 7

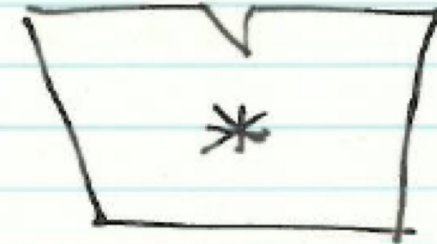
end of cycle 7:

	V	flag	value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	Y	~

a	0	X	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	Y	~



x	1	~	1	1	~	2
y	0	b	~	0	c	~



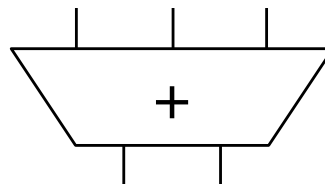
- \* All 6 instructions renamed.
- Note what happened to R5

# State of RAT and RS in Cycle 7

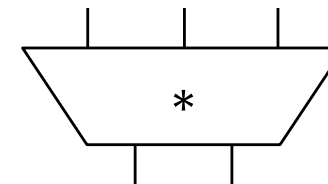
	Cycle	1	2	3	4	5	6	7
MUL R1, R2 → R3	F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	
ADD R3, R4 → R5		F	D	-	-	-	-	
ADD R2, R6 → R7			F	D	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	
ADD R8, R9 → R10				F	D	E <sub>1</sub>	E <sub>2</sub>	
MUL R7, R10 → R11					F	D	-	
ADD R5, R11 → R5						F	D	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



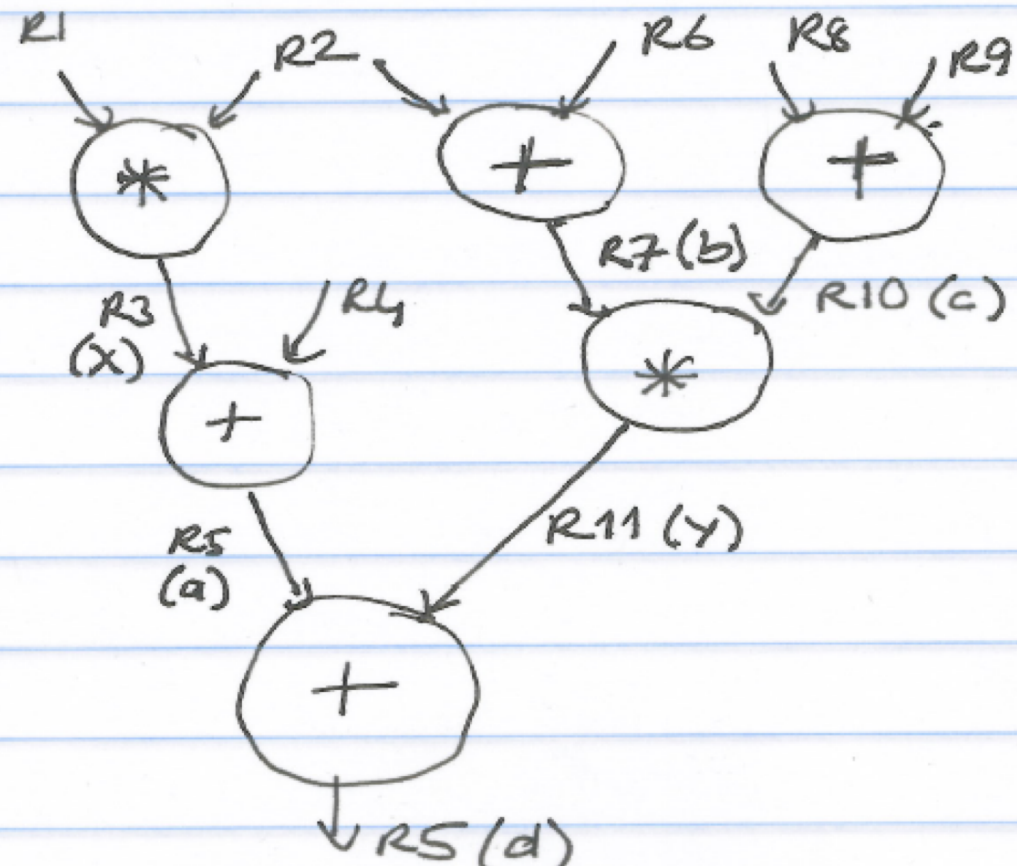
# Corresponding Dataflow Graph (Reverse Engineered)

MUL R1, R2 → R3 (x)  
ADD R3, R4 → R5 (a)  
ADD R2, R6 → R7 (b)  
ADD R8, R9 → R10 (c)  
MUL R7, R10 → R11 (y)  
ADD R5, R11 → R5 (d)

## Dataflow graph

Nodes: operations performed by the instructions

Arcs: tags in Tomasulo's algorithm





# Some More Questions (Design Choices)

---

- When is a reservation station entry deallocated?
- Exactly when does an instruction broadcast its tag?
- Should the reservation stations be dedicated to each functional unit or global across functional units?
  - Centralized vs. Distributed: What are the tradeoffs?
- Should reservation stations and ROB store data values or should there be a centralized physical register file where all data values are stored?
  - What are the tradeoffs?
- Many other design choices for OoO engines

# For You: An Exercise, w/ Precise Exceptions

---

MUL R3  $\leftarrow$  R1, R2  
ADD R5  $\leftarrow$  R3, R4  
ADD R7  $\leftarrow$  R2, R6  
ADD R10  $\leftarrow$  R8, R9  
MUL R11  $\leftarrow$  R7, R10  
ADD R5  $\leftarrow$  R5, R11



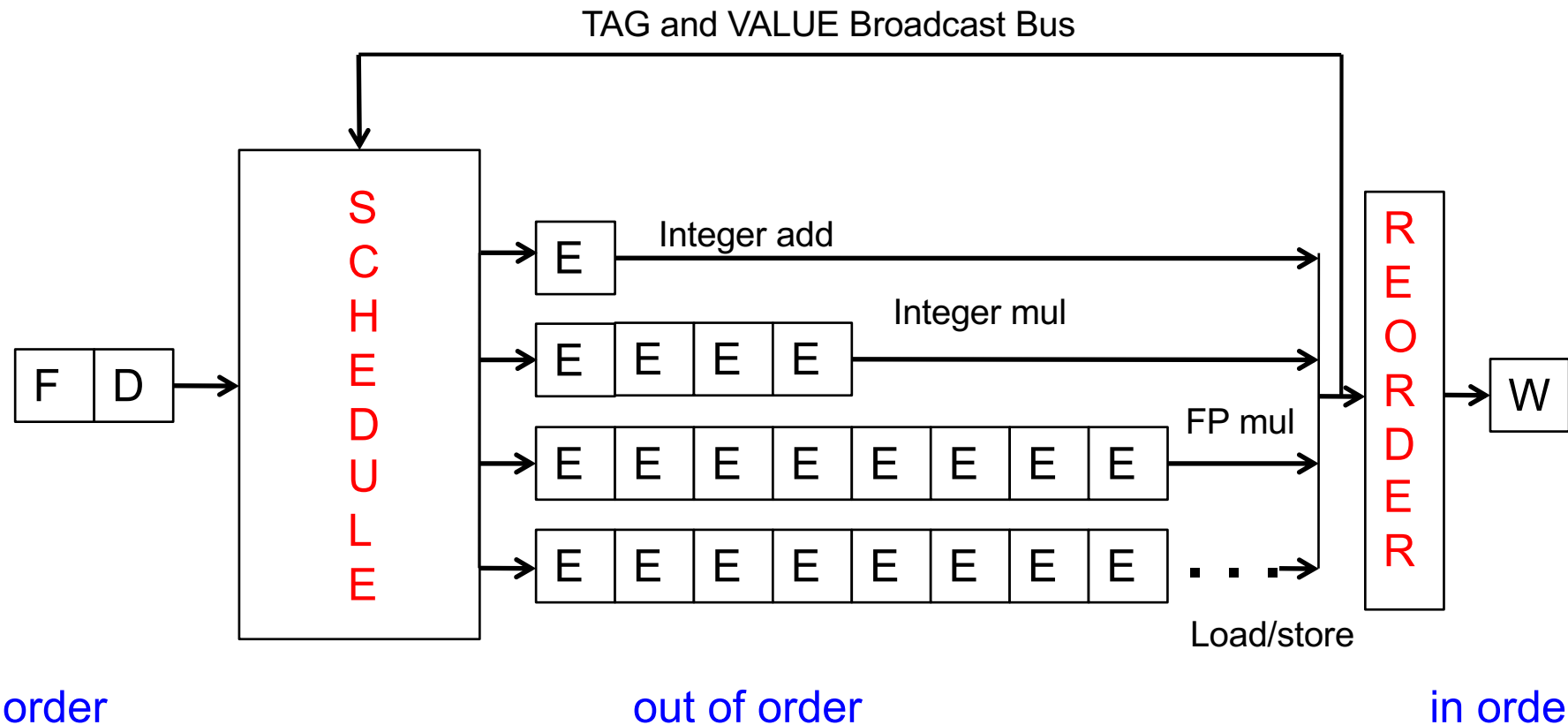
- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with reorder buffer (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine with reorder buffer (full forwarding)

# Out-of-Order Execution with Precise Exceptions

---

- **Idea:** Use a reorder buffer to reorder instructions before committing them to architectural state
- An instruction updates the RAT when it completes execution
  - Also called **frontend register file**
- An instruction updates a **separate architectural register file** when it retires
  - i.e., when it is the oldest in the machine and has completed execution
  - In other words, **the architectural register file is always updated in program order**
- On an exception: flush pipeline, copy architectural register file into frontend register file

# Out-of-Order Execution with Precise Exceptions



- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

# Two Humps in a Modern Pipeline

---

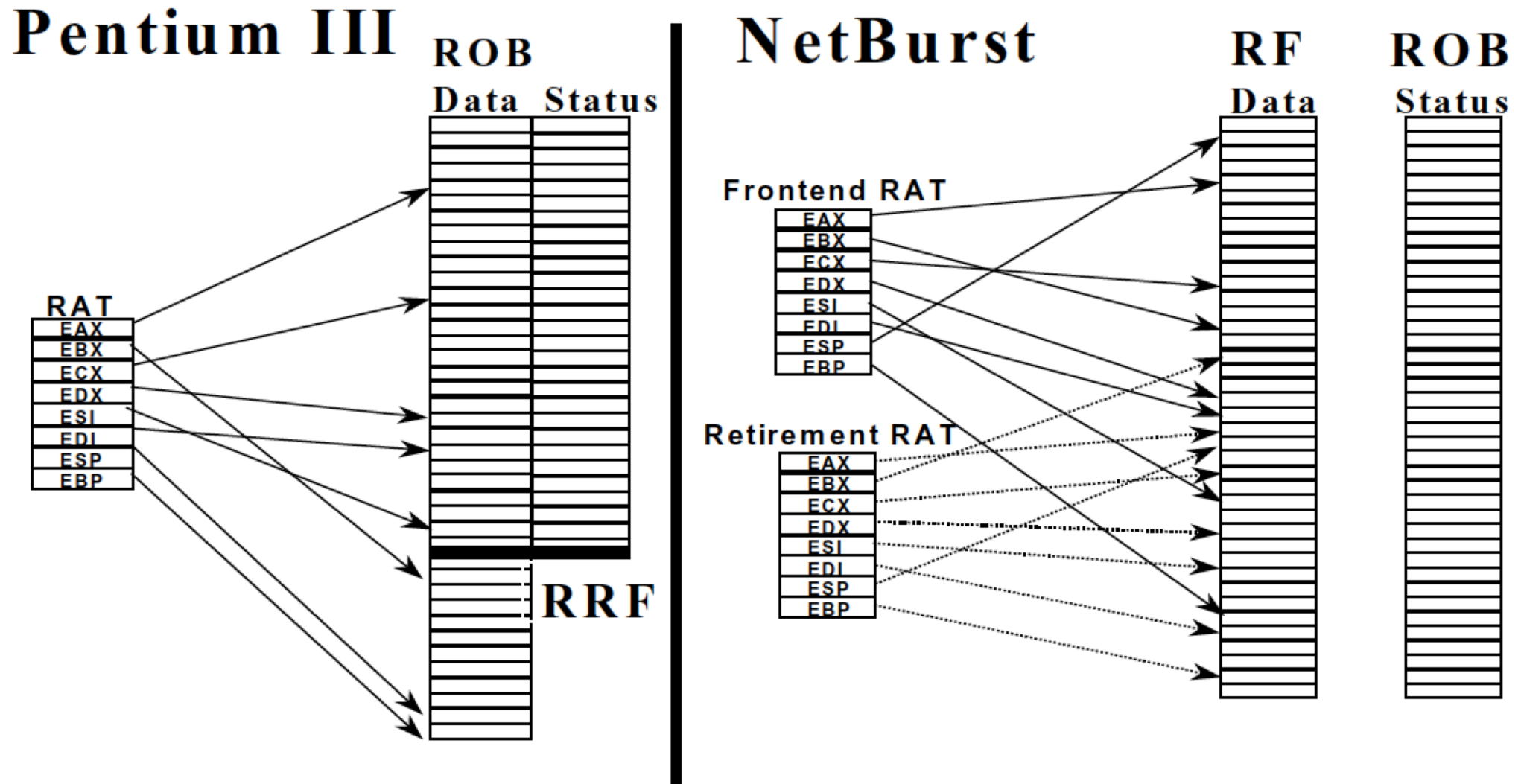


# Modern OoO Execution w/ Precise Exceptions

---

- Most modern processors use the following
  - **Reorder buffer** to support in-order retirement of instructions
  - **A single register file** to store all registers
    - Both speculative and architectural registers
    - INT and FP are still separate
  - **Two register maps**
    - Future/frontend register map → used for renaming
    - Architectural register map → used for maintaining precise state

# An Example from Modern Processors



Boggs et al., "The Microarchitecture of the Pentium 4 Processor,"  
Intel Technology Journal, 2001.



# Enabling OoO Execution, Revisited

---

1. **Link** the consumer of a value to the producer
  - **Register renaming**: Associate a “tag” with each data value
2. **Buffer** instructions until they are ready
  - Insert instruction into **reservation stations** after renaming
3. Keep **track** of **readiness** of source values of an instruction
  - **Broadcast the “tag”** when the value is produced
  - Instructions **compare their “source tags”** to the broadcast tag  
→ if match, source value becomes ready
4. When all source values of an instruction are ready, **dispatch** the instruction to functional unit (FU)
  - **Wakeup and select/schedule** the instruction



# Summary of OOO Execution Concepts

---

- Register renaming eliminates false dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch

# OOO Execution: Restricted Dataflow

---

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
- The dataflow graph is limited to the **instruction window**
  - Instruction window: all decoded but not yet retired instructions
- Can we do it for the whole program?
- Why would we like to?
- In other words, how can we have a large instruction window?
- Can we do it efficiently with Tomasulo's algorithm?

# Recall: Dataflow Graph for Our Example

---

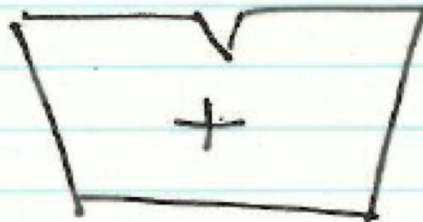
MUL R3  $\leftarrow$  R1, R2  
ADD R5  $\leftarrow$  R3, R4  
ADD R7  $\leftarrow$  R2, R6  
ADD R10  $\leftarrow$  R8, R9  
MUL R11  $\leftarrow$  R7, R10  
ADD R5  $\leftarrow$  R5, R11

# Recall: State of RAT and RS in Cycle 7

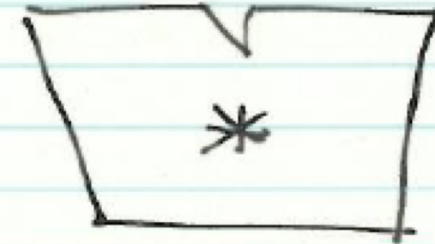
end of cycle 7:

	V	tag	value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	Y	~

a	0	X	~	1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a	~	0	Y	~



x	1	~	1	1	~	2
y	0	b	~	0	c	~



- \* All 6 instructions renamed.
- Note what happened to R5

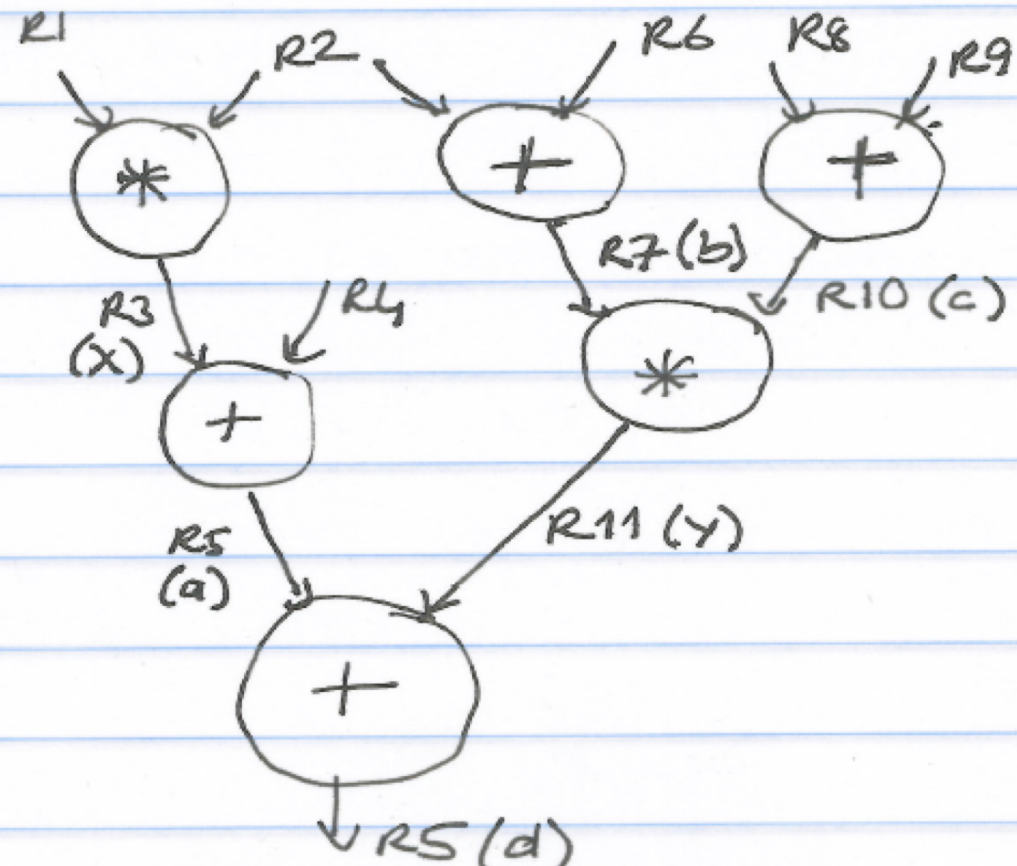
# Recall: Dataflow Graph

MUL R1, R2  $\rightarrow$  R3 (x)  
ADD R3, R4  $\rightarrow$  R5 (a)  
ADD R2, R6  $\rightarrow$  R7 (b)  
ADD R8, R9  $\rightarrow$  R10 (c)  
MUL R7, R10  $\rightarrow$  R11 (y)  
ADD R5, R11  $\rightarrow$  R5 (d)

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm

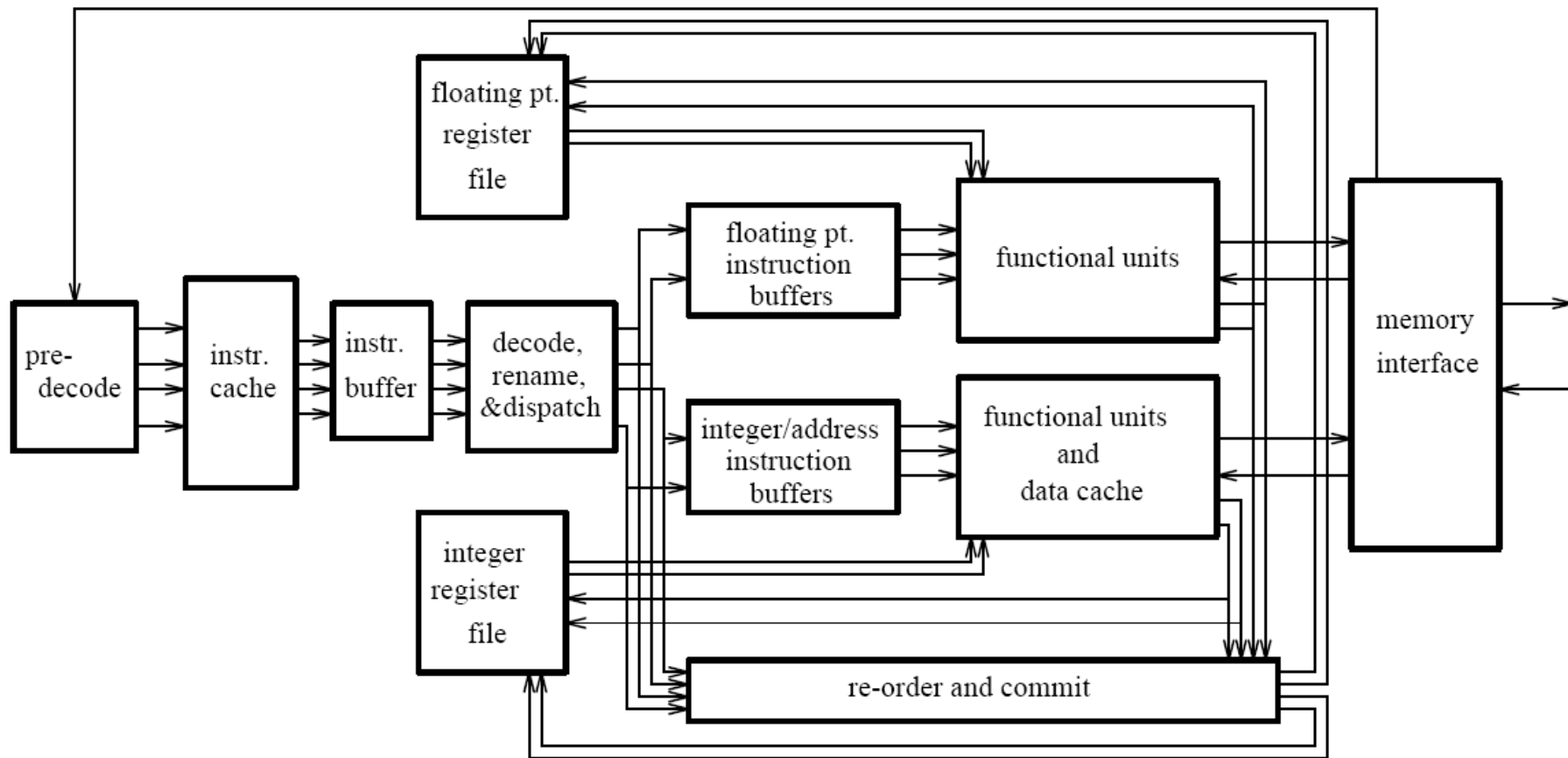


# Questions to Ponder

---

- Why is OoO execution beneficial?
  - What if all operations take a single cycle?
  - **Latency tolerance**: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently
  
- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - **What limits the latency tolerance scalability of Tomasulo's algorithm?**
    - **Active/instruction window size**: determined by both scheduling window and reorder buffer size

# General Organization of an OOO Processor



- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proc. IEEE, Dec. 1995.



# A Modern OoO Design: Intel Pentium 4

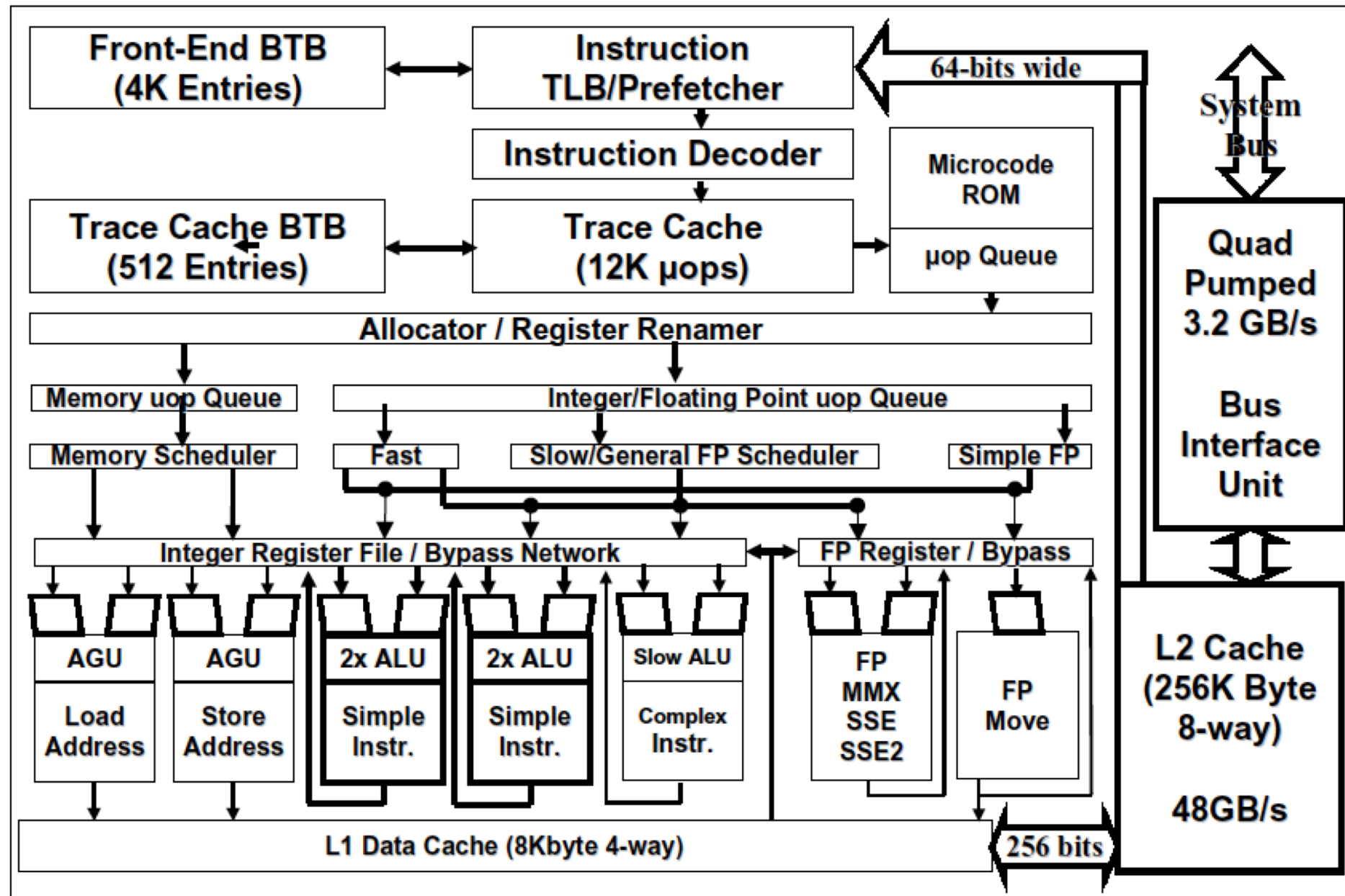
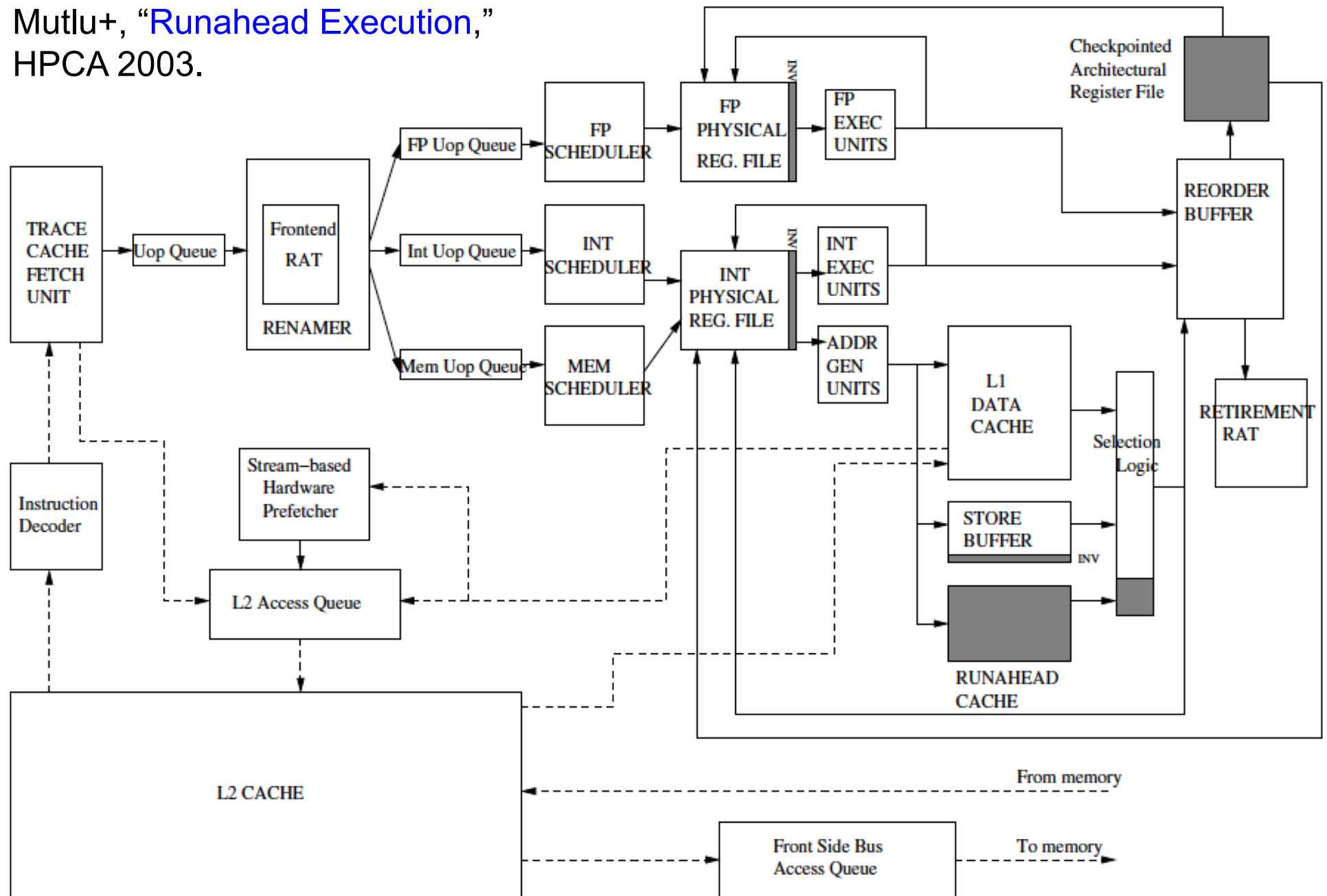


Figure 4: Pentium® 4 processor microarchitecture



# Intel Pentium 4 Simplified

Mutlu+, “Runahead Execution,”  
HPCA 2003.



# Alpha 21264

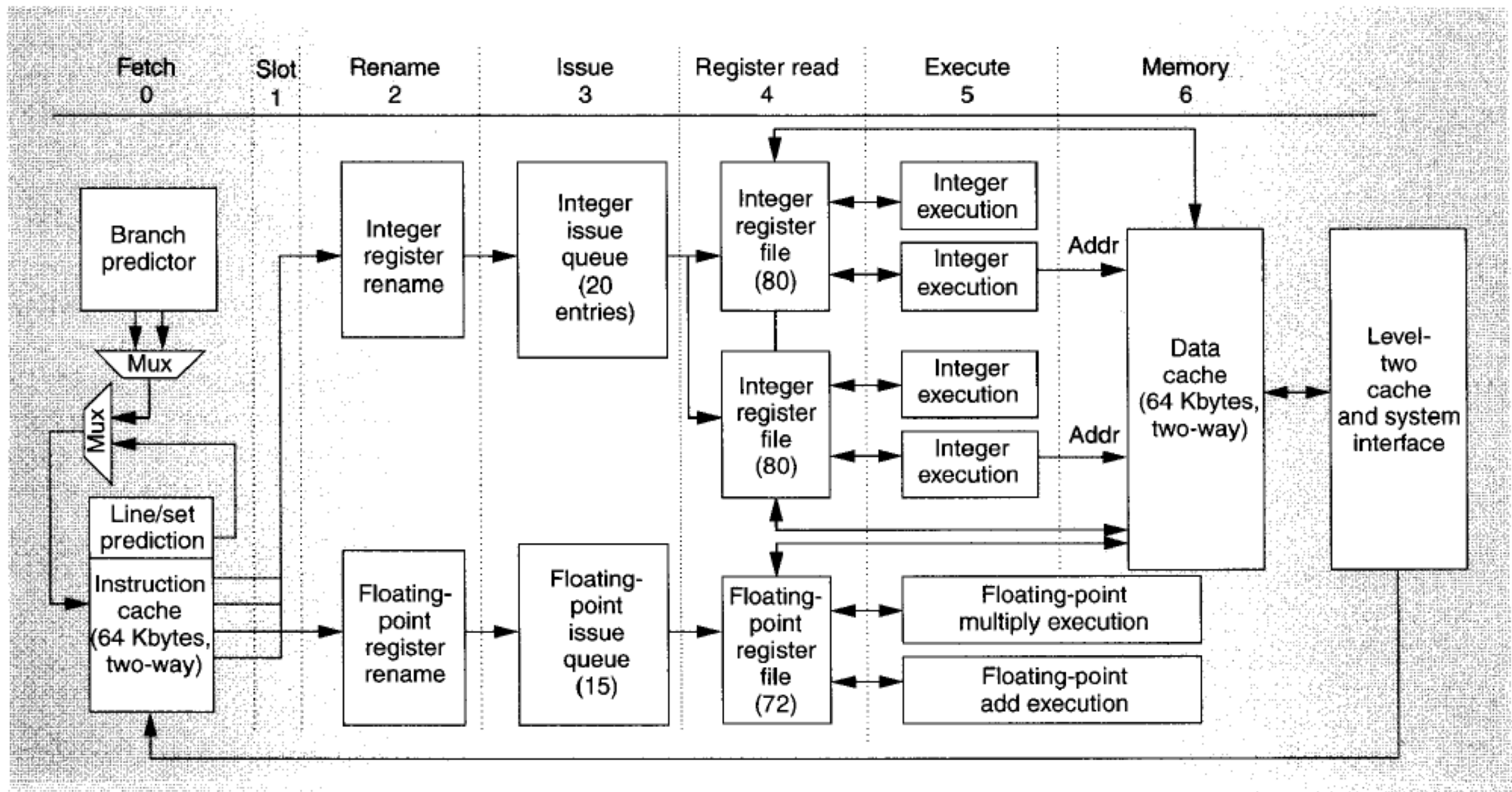
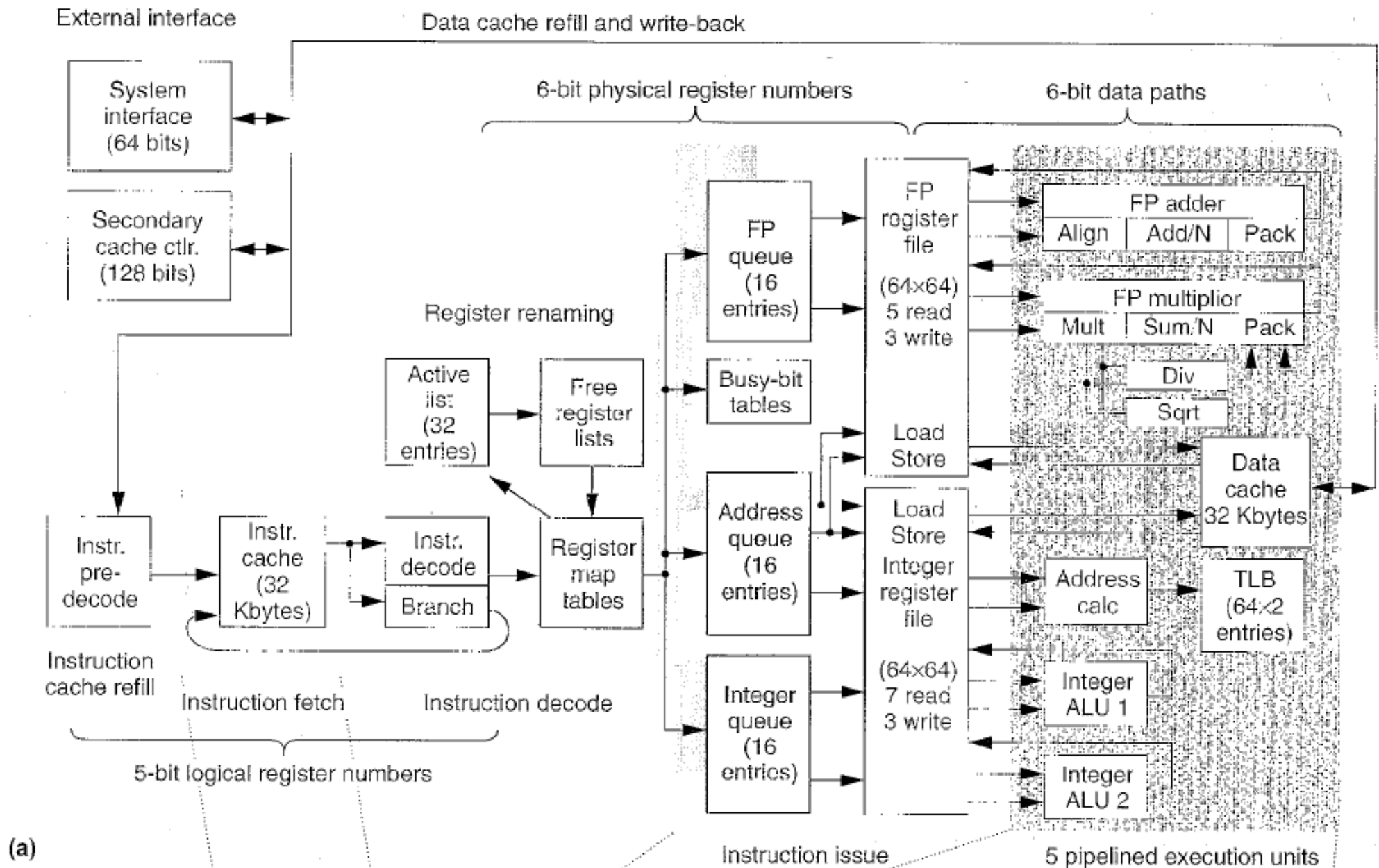


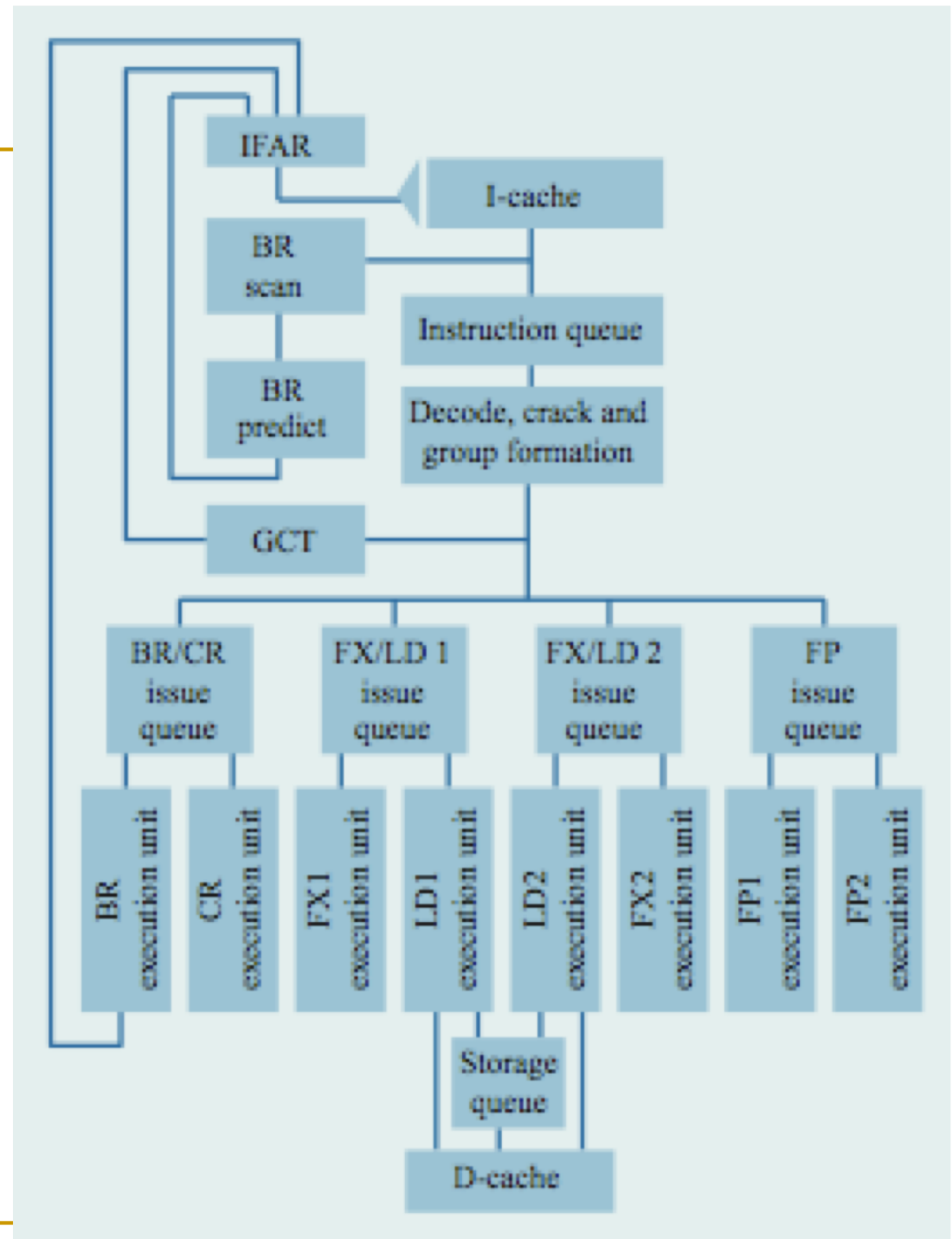
Figure 2. Stages of the Alpha 21264 instruction pipeline.

# MIPS R10000



# IBM POWER4

- Tendler et al.,  
“POWER4 system  
microarchitecture,”  
IBM J R&D, 2002.



# IBM POWER4

---

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching

# IBM POWER5

- Kalla et al., “IBM Power5 Chip: A Dual-Core Multithreaded Processor,” IEEE Micro 2004.

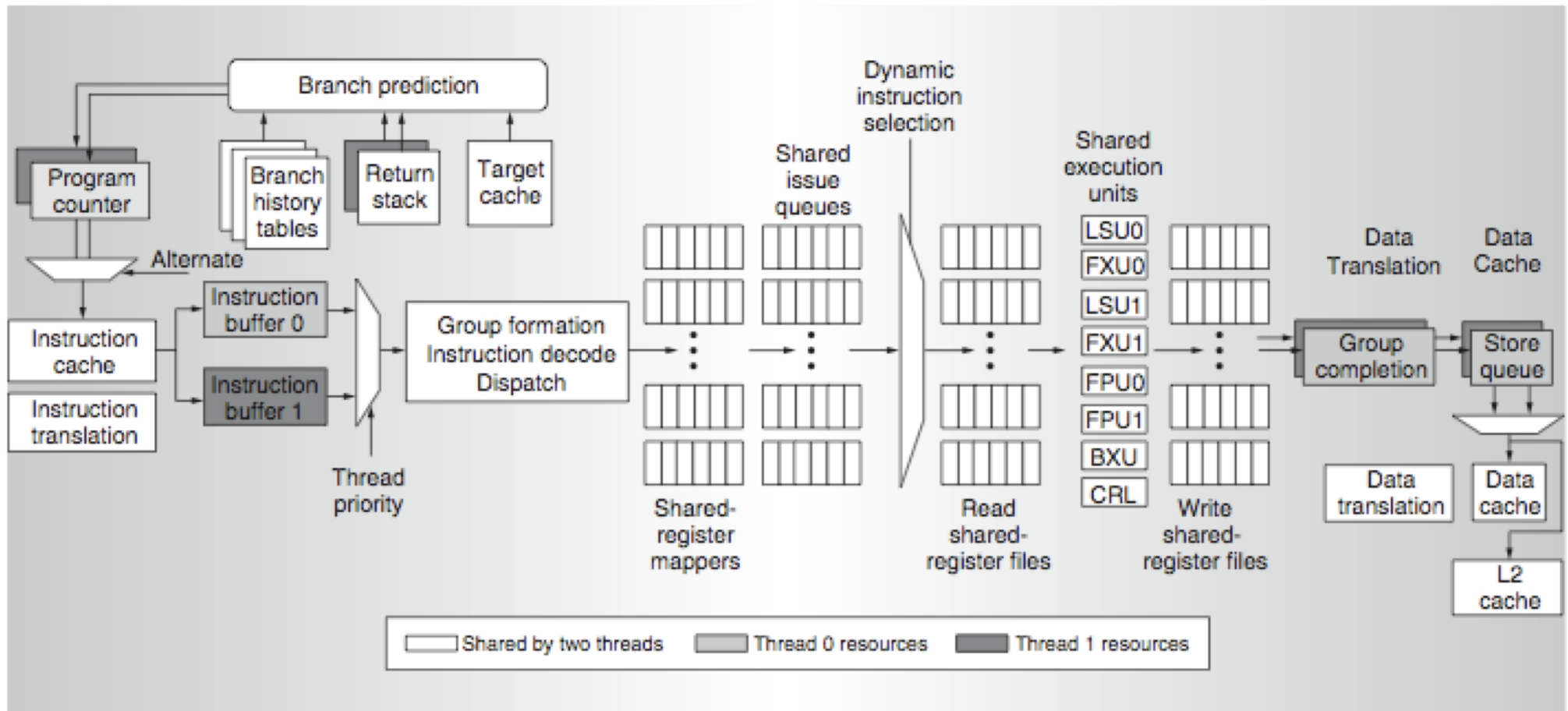


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

# Handling Out-of-Order Execution of Loads and Stores



# Registers versus Memory

---

- So far, we considered mainly registers as part of state
- What about memory?
- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)



# Memory Dependence Handling (I)

---

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance
- Observation and Problem: Memory address is not known until a load/store executes
- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores need to be handled *after* their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine

# Memory Dependence Handling (II)

---

- When do you schedule a load instruction in an OOO engine?
  - Problem: A younger load can have its address ready before an older store's address is known
  - Known as the **memory disambiguation** problem or the **unknown address** problem
  
- Approaches
  - **Conservative:** Stall the load until all previous stores have computed their addresses (or even retired from the machine)
  - **Aggressive:** Assume load is independent of unknown-address stores and schedule the load right away
  - **Intelligent:** Predict (with a more sophisticated predictor) if the load is dependent on the/any unknown address store

# Handling of Store-Load Dependences

---

- **A load's dependence status is not known** until all previous store addresses are available.
- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check for address match)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address
- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load dependent on all previous stores
  - Option 2: Assume load independent of all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

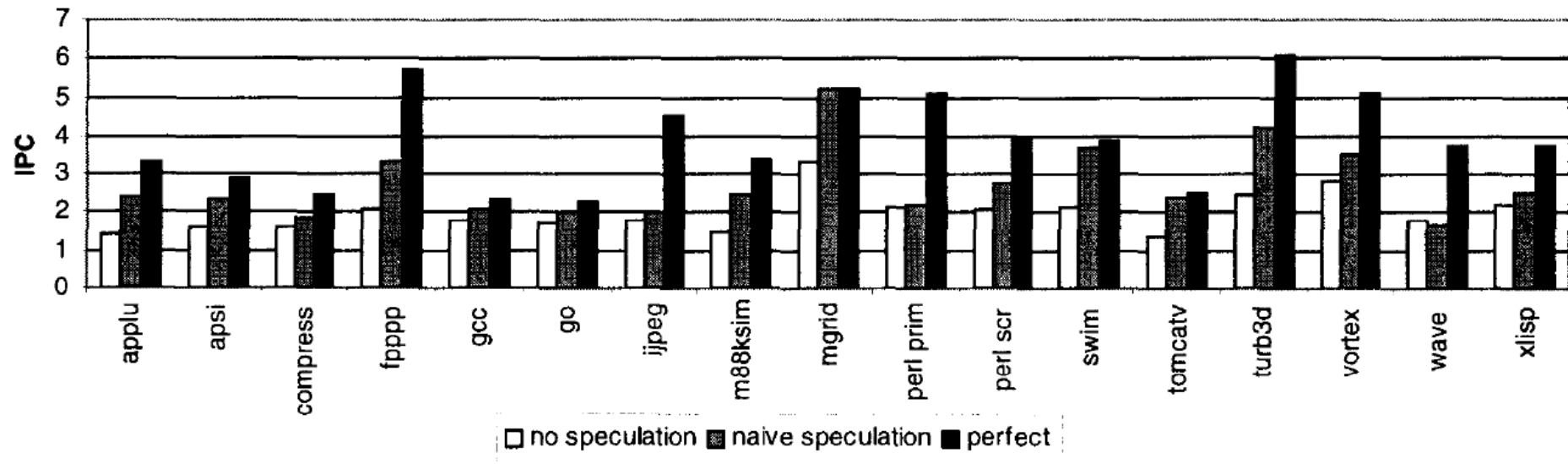
# Memory Disambiguation (I)

---

- Option 1: Assume load is dependent on all previous stores
  - + No need for recovery
  - Too conservative: delays independent loads unnecessarily
- Option 2: Assume load is independent of all previous stores
  - + Simple and can be common case: no delay for independent loads
  - Requires recovery and re-execution of load and dependents on misprediction
- Option 3: Predict the dependence of a load on an outstanding store
  - + More accurate. Load store dependencies persist over time
  - Still requires recovery/re-execution on misprediction
    - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
    - Moshovos et al., “**Dynamic speculation and synchronization of data dependences**,” ISCA 1997.
    - Chrysos and Emer, “**Memory Dependence Prediction Using Store Sets**,” ISCA 1998.

# Memory Disambiguation (II)

- Chrysos and Emer, “Memory Dependence Prediction Using Store Sets,” ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

# Data Forwarding Between Stores and Loads

---

- We cannot update memory out of program order  
→ Need to buffer all store and load instructions in instruction window
- Even if we **know** all addresses of past stores when we generate the address of a load, two questions still remain:
  1. How do we check whether or not it is dependent on a store
  2. How do we forward data to the load if it is dependent on a store
- Modern processors use a LQ (load queue) and an SQ for this
  - Can be combined or separate between loads and stores
  - A load searches the SQ after it computes its address. Why?
  - A store searches the LQ after it computes its address. Why?

# Out-of-Order Completion of Memory Ops

---

- When a store instruction finishes execution, it writes its address and data in its reorder buffer entry
- When a later load instruction generates its address, it:
  - searches the reorder buffer (or the SQ) with its address
  - accesses memory with its address
  - receives the value from the youngest older instruction that wrote to that address (either from ROB or memory)
- This is a complicated “search logic” implemented as a Content Addressable Memory
  - Content is “memory address” (but also need *size* and *age*)
  - Called **store-to-load forwarding logic**

# Store-Load Forwarding Complexity

---

- **Content Addressable Search** (based on Load Address)
- **Range Search** (based on Address and Size of both the Load and earlier Stores)
- **Age-Based Search** (for last written values)
- **Load data can come from a combination of multiple places**
  - ❑ One or more stores in the Store Buffer (SQ)
  - ❑ Memory/cache



# Other Approaches to Concurrency (or Instruction Level Parallelism)

# Approaches to (Instruction-Level) Concurrency

---

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

# Review: Data Flow: Exploiting Irregular Parallelism

# Data Flow Summary

---

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)
- Data Flow at the ISA level has not been (as) successful
- Data Flow implementations under the hood (while preserving sequential ISA semantics) have been very successful
  - Out of order execution is the prime example

# Pure Data Flow Advantages/Disadvantages

---

## ■ Advantages

- ❑ Very good at exploiting **irregular parallelism**
- ❑ Only real dependencies constrain processing
- ❑ More parallelism can be exposed than von Neumann model

## ■ Disadvantages

- ❑ No precise state semantics
  - Debugging very difficult
  - Interrupt/exception handling is difficult (what is precise state semantics?)
- ❑ Too much parallelism? (Parallelism control needed)
- ❑ High bookkeeping overhead (tag matching, data storage)
- ❑ ...

# Approaches to (Instruction-Level) Concurrency

---

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

# Superscalar Execution

# Superscalar Execution

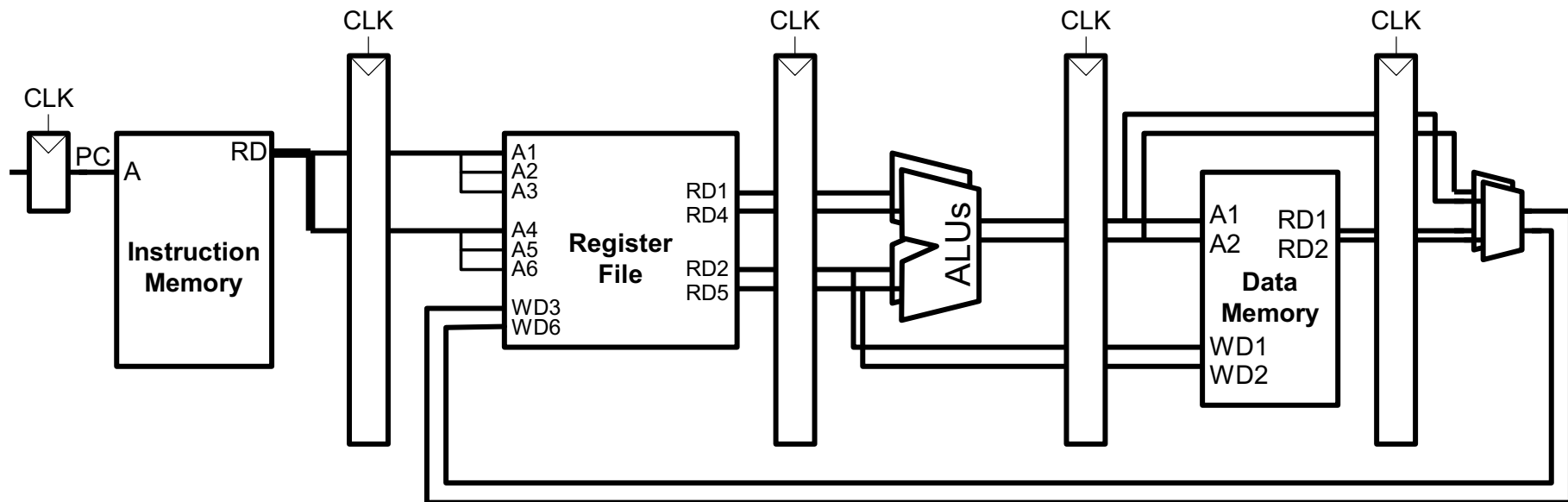
---

- Idea: Fetch, decode, execute, retire **multiple instructions per cycle**
  - N-wide superscalar → N instructions per cycle
- Need to add the hardware resources for doing so
- Hardware performs the dependence checking between concurrently-fetched instructions
- Superscalar execution and out-of-order execution are orthogonal concepts
  - Can have all four combinations of processors:  
[in-order, out-of-order] x [scalar, superscalar]



# In-Order Superscalar Processor Example

- Multiple copies of datapath: Can issue multiple instructions at per cycle
- Dependencies make it tricky to issue multiple instructions at once

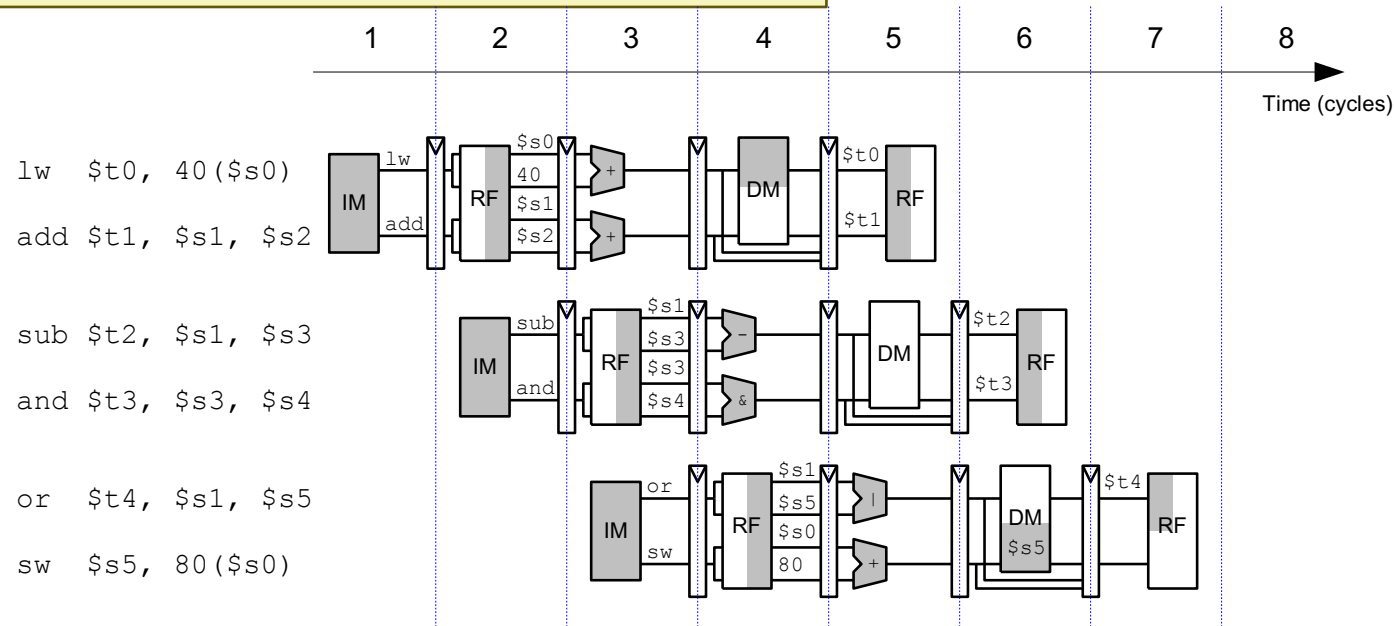


*Here: Ideal IPC = 2*

# In-Order Superscalar Performance Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

*Ideal IPC = 2*

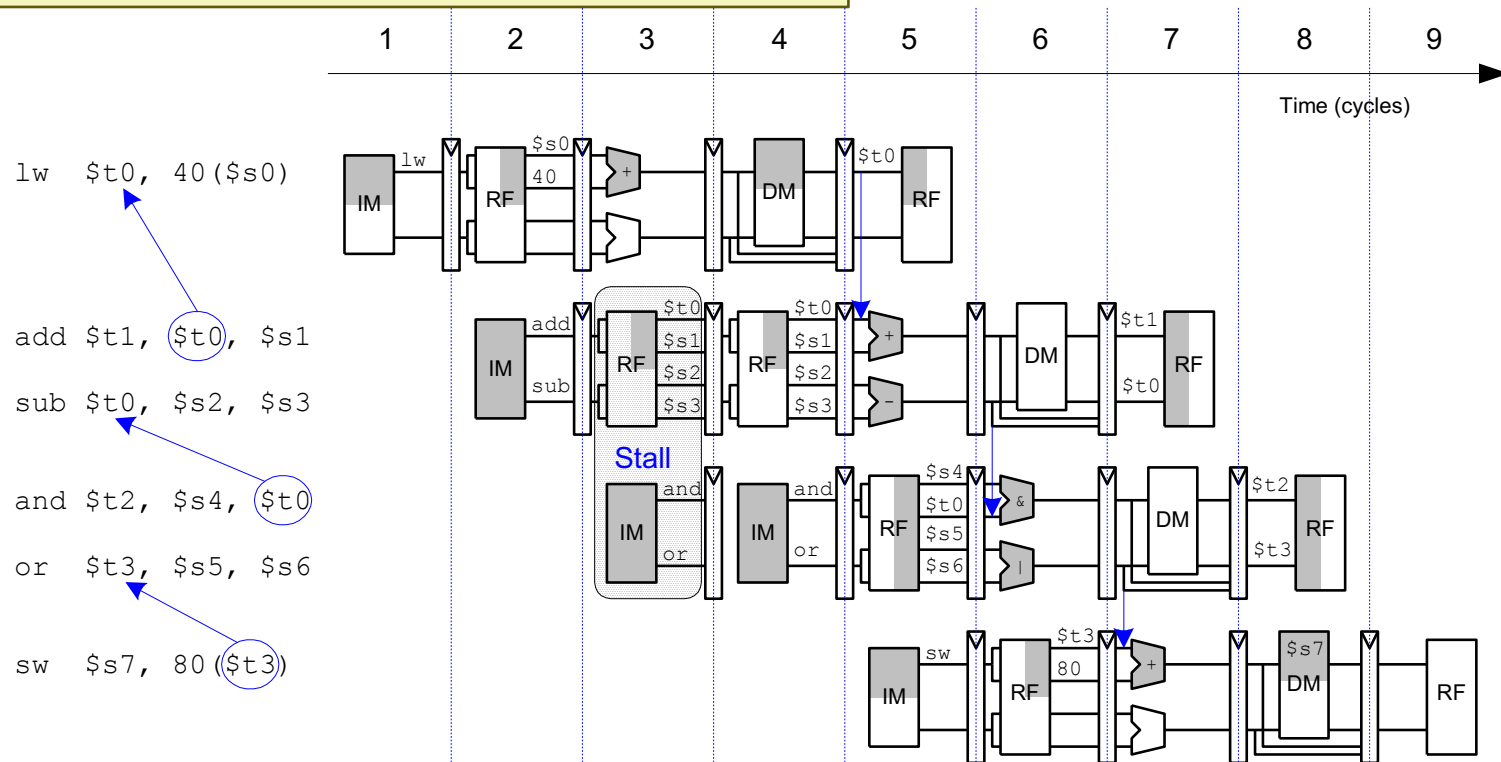


*Actual IPC = 2* (6 instructions issued in 3 cycles)

# Superscalar Performance with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

*Ideal IPC = 2*



*Actual IPC = 1.2* (6 instructions issued in 5 cycles)

# Superscalar Tradeoffs

---

- Advantages

- Higher IPC (instructions per cycle)

- Disadvantages

- Higher complexity for dependency checking
  - Require checking within a pipeline stage
  - Renaming becomes more complex in an OoO processor
- More hardware resources needed